A Practical Guide to Getting Started with Together ControlCenter

This set of pages is an introductory tutorial and guide for new Together users. Upon completing the tutorial, you should be familiar enough with Together to create applications from scratch. Plan to spend several hours going through this initial material. For a more in-depth introduction you may want to take longer.

Contents:

Using the Practical Guide

Together can be described in many ways. It is a class modeling tool, always keeping source and model diagrams in sync. It is an architectural guide, revealing the physical and logical layout of a project. It is the primary communication link among analysts, designers, developers, and programmers. It is a customizable Java and C++ programming environment, with features promoting the best practices in software development. It is an enterprise application development enabler ... and much, much more.

Together is user-friendly but feature rich. This Practical Guide is a collection of pages designed to give you a quick, working knowledge of Together. The collection is divided into two major sections:

1. Quick Tour -- shows how to navigate your way around Together.
2. Tutorial -- shows how to construct a project from scratch.

You should go through the Quick Tour first -- take advantage of the EXERCISE buttons. We strongly recommend that you do the exercises or at least do some significant exploration on your own.

When you have learned how to navigate your way around Together, you're ready for the Tutorial. It is built around many steps, formatted as follows.

Step: Something for you to do.

Each step is followed by an explanation, with how-to snapshots and related information.

We can't begin to tell you everything about Together, but this will get you started. You will discover lots of features not covered in this tutorial when you use Together for your own work. Have fun!

Quick Tour: Navigating your way around Together

The Quick Tour shows the layout and functionality of the Together user interface. It gives insight into the way you can customize Together for your own work.

1. The Main Window

Tutorial: Building a Together project from scratch

The Tutorial covers the basic features of Together by leading you through the steps of creating a Java application. The Together Tutorial is in Java, although C++ programmers could easily mimic many of the steps in C++ instead of Java. (At most only a minimal knowledge of Java is required.)

Software requirements and additional resources

The Practical Guide requires no special software beyond Together. Together Solo is sufficient for most of the work, although you'll need Together ControlCenter for some advanced features.

The Together product line, including Together ControlCenter and Together Solo can be downloaded at www.togethersoft.com. Together ControlCenter, the more feature-laden of the Together products, is available for a free 15-day trial. An abbreviated version of Together Solo is available for free from www.togethercommunity.com.

Together software includes a complete set of documentation. Together users have a forum for exchanging practical information at the Together Community site, www.togethercommunity.com.

Last Revised: Thu, Apr 19, 2001

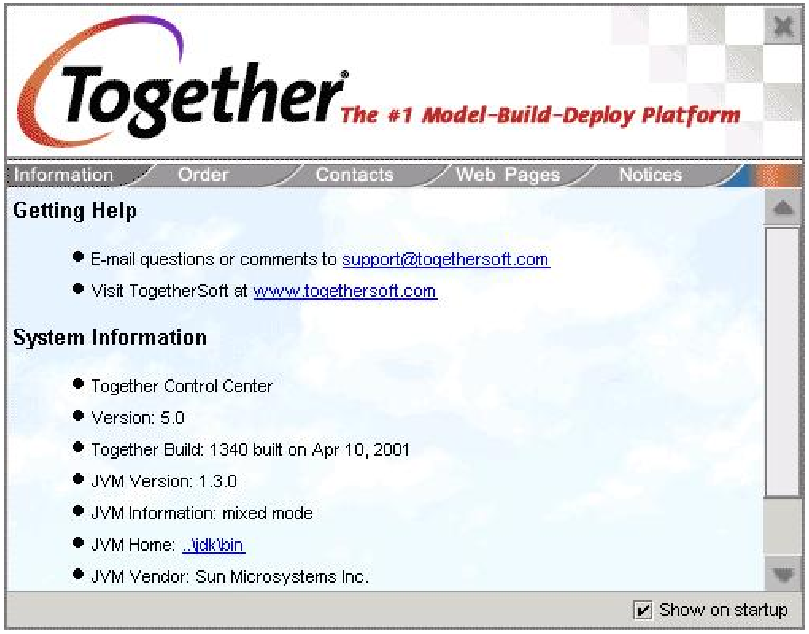
Together Quick Tour
Part 1: The Main Window

The Quick Tour navigates through some of the basic features of Together. You will start the tour by taking a good look at Together's Main window. The **CashSales** sample project will form the basis for many of the tour discussions.

Contents

Opening a project

When you open Together for the first time, it displays an "about" splash screen in the middle of its window. You'll need to close the splash screen before Together will respond to any commands. Click the X in the upper right corner.



Getting Help

- E-mail questions or comments to support@togethersoft.com
- Visit TogetherSoft at www.togethersoft.com

System Information

- Together Control Center
- Version: 5.0
- Together Build: 1340 built on Apr 10, 2001
- JVM Version: 1.3.0
- JVM Information: mixed mode
- JVM Home: ..\jdk\bin
- JVM Vendor: Sun Microsystems Inc.

Together works almost entirely within the context of projects. When you open Together without a project, the only information Together shows is in its Explorer pane on the left side of the Main window.

The Explorer pane displays both physical and logical organizations of files. You can use it to navigate within the physical directory of the system or within a project.

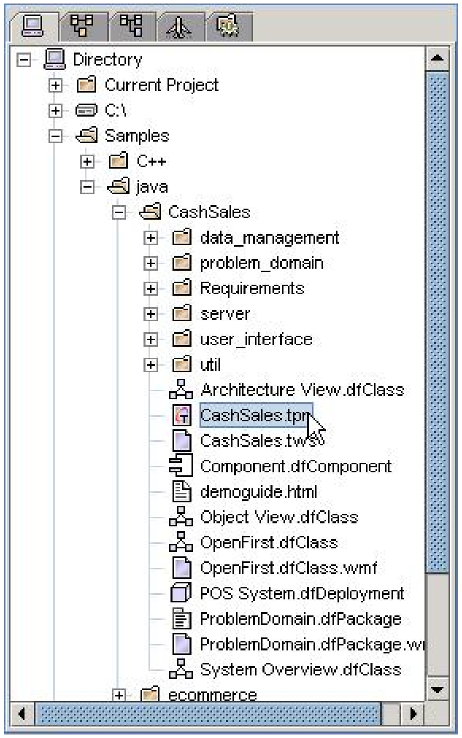
The Explorer pane is organized by tabs. The Directory tab (left most tab) shows both the physical directory of your system and the organization of the Together home folder with respect to projects.

CashSales is a Together sample project that models a simple retail store cash transaction. You can select the project in the Directory by first expanding **Samples**, then **java**, then **CashSales**.

The Directory displays an icon beside a file name to indicate its type with respect to Together.

- files that can be opened with Together's editor
- Together project files
- other types of files that cannot be opened in the editor

Double-click on **CashSales.tpr** to open the **CashSales** project.



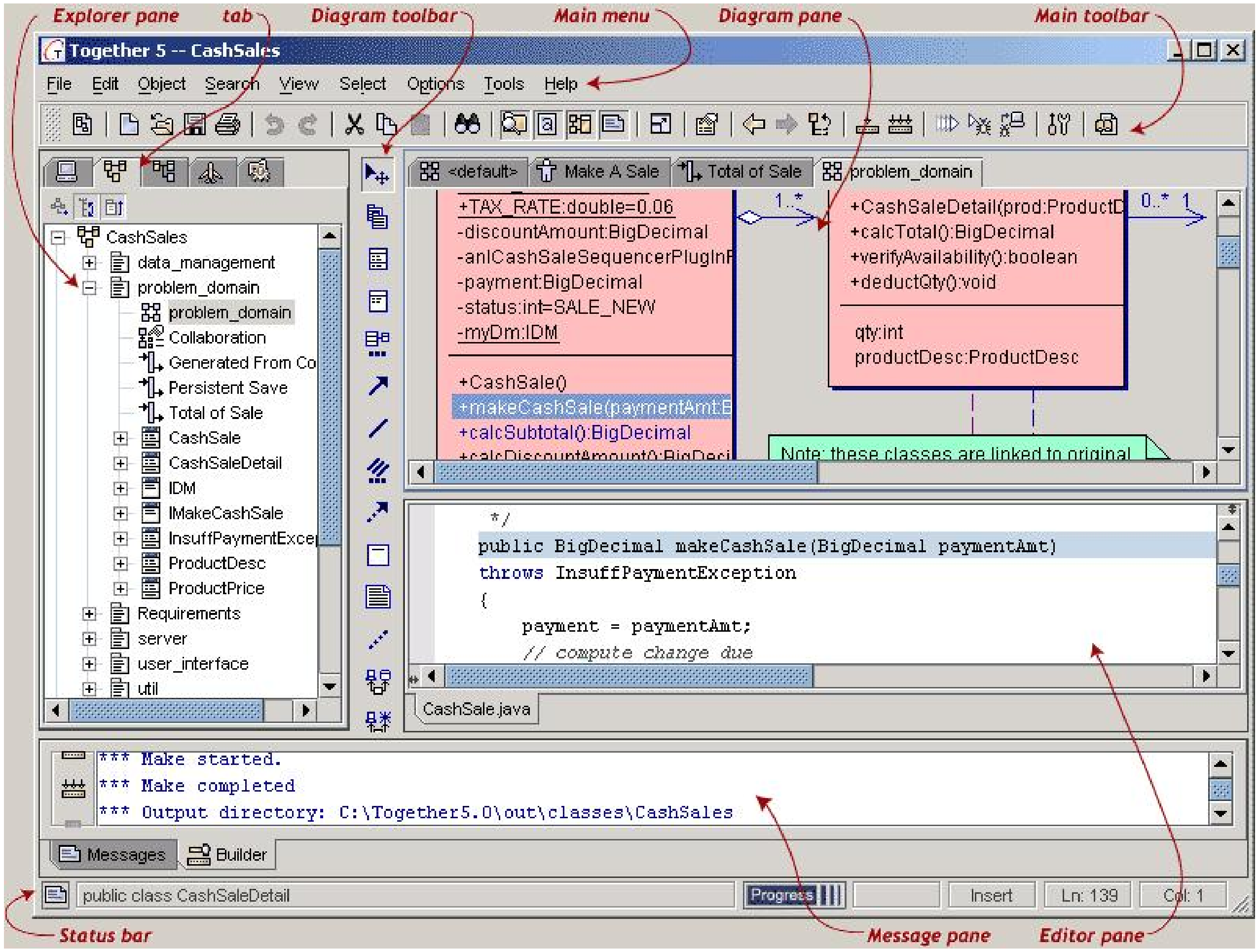
```
Directory
  Current Project
  C:\
  Samples
    C++
    java
      CashSales
        data_management
        problem_domain
        Requirements
        server
        user_interface
        util
        Architecture View.dfClass
        CashSales.tpr
        CashSales.tws
        Component.dfComponent
        demoguide.html
        Object View.dfClass
        OpenFirst.dfClass
        OpenFirst.dfClass.wmf
        POS System.dfDeployment
        ProblemDomain.dfPackage
        ProblemDomain.dfPackage.wı
        System Overview.dfClass
      ecommerce
```

EXERCISE

Understanding the Main window organization

When you open **CashSales**, the Main window divides into panes. Together has four major panes.

	Explorer pane	for system navigation
	Editor pane	for viewing and editing source code and ordinary text
	Diagram pane	for creating UML and other kinds of model diagrams
	Message pane	for system messages and tasks



```
    */
public BigDecimal makeCashSale(BigDecimal paymentAmt)
throws InsuffPaymentException
{
    payment = paymentAmt;
    // compute change due
```

```
*** Make started.
*** Make completed
*** Output directory: C:\Together5.0\out\classes\CashSales
```

If a pane is hidden, click its view button on the Main toolbar. The right most button (⊡) is a toggle to expand the current pane (the pane with the light blue border) to fill the entire window. (In the snapshot above, the current pane is the Diagram pane.)

Each pane has tabs for the page in focus. Clicking on a tab brings its page into focus.

You can resize the panes by moving the separators between them.

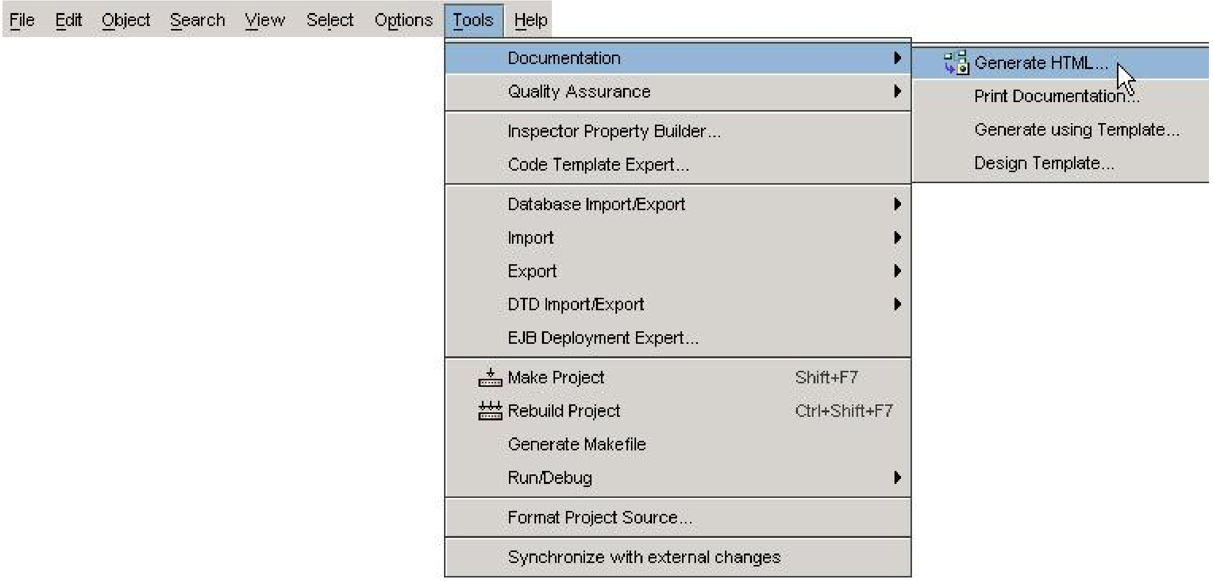
EXERCISE

Exploring the Main menu

The Main menu has nine commands.

File	Project and file operations
Edit	Editing operations plus "infinite" undo/redo (for most operations -- not just editor changes)
Object	Context sensitive menu whose operations vary according to the currently selected object. (This item is available only if the "current item" has a speedmenu.)
Search	Search and replace across multiple files
View	Toggle panes between hidden and displayed
Select	Navigate among panes and diagrams
Options	Customize your Together configuration
Tools	Access to several system modules
Help	Hypertext documentation for Together

This snapshot below shows the **Tools** menu along with its Documentation submenu.

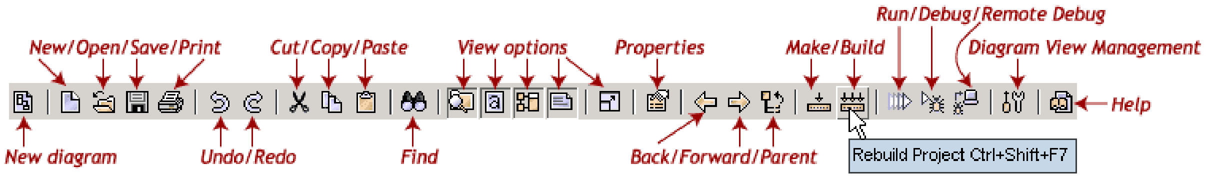


Most of the Main menu commands have similar cascading menus. Some of the selections have keyboard equivalents. For example, the **File|Open Project** keyboard equivalent is Ctrl+Shift+O.

EXERCISE

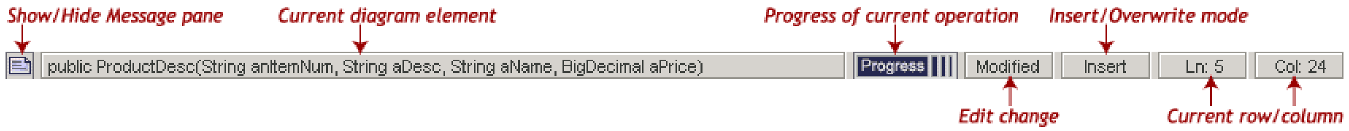
Exploring the Main toolbar and status bar

Buttons on the Main toolbar correspond to some of the commonly used commands from the Main menu.



Each toolbar button has a flyover, which pops up on a mouse-over. The illustration above shows the flyover for the button "Rebuild Project." Incidentally, the Main toolbar is undockable -- simply drag it off the window.

The status bar is at the bottom of the Main window. The large box on the status bar changes according to the mouse-over on the Diagram pane.



public ProductDesc(String anItemNum, String aDesc, String aName, BigDecimal aPrice)

Most items on the status bar are self-explanatory, except perhaps the "Diagram View Management" button. That button pops up the Diagram Options window to let you show or hide different kinds of diagram content.

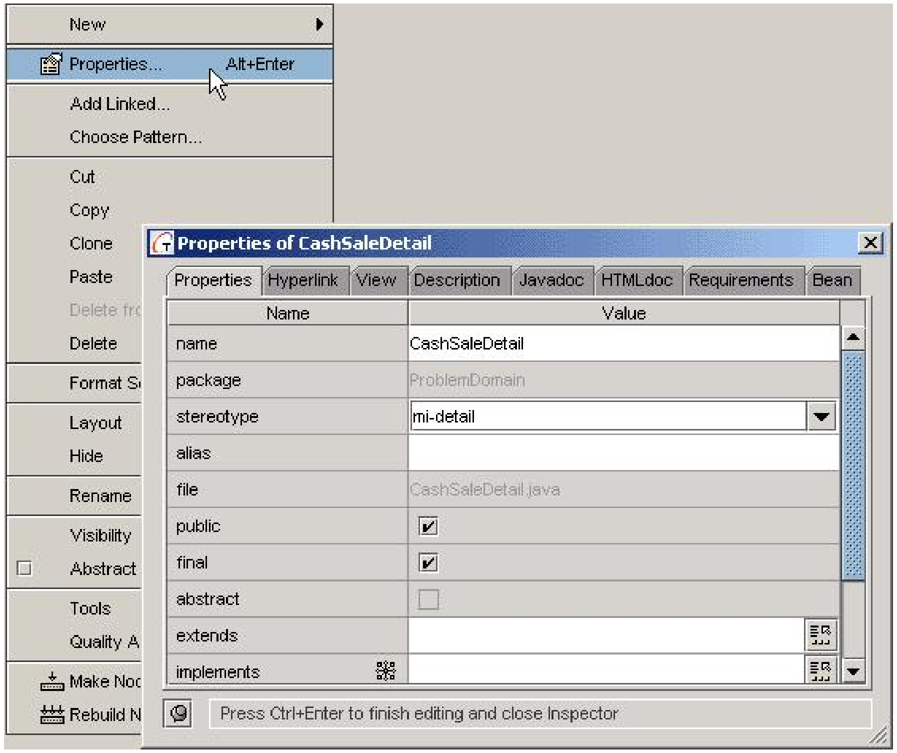
EXERCISE

Accessing speedmenus and properties inspectors

Most elements of the Together user interface have speedmenus, also known as "context" or "right-click" menus. They give quick access to common element tasks. (Elements include Together artefacts such as diagrams as well as UML elements such as class nodes, associations, and use cases.) Many elements also have properties inspectors for easy extensive customization.

Most speedmenus list properties inspectors among their choices. The snapshot below on the shows a class speedmenu in the back with a properties inspector superimposed in front.

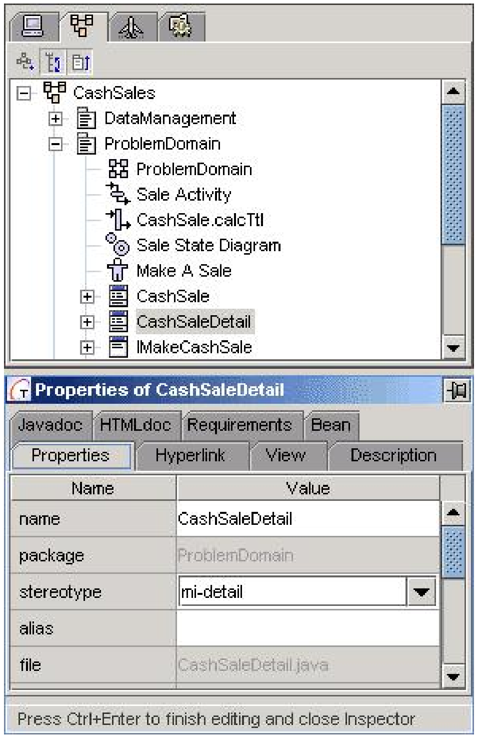
To get to a class speedmenu, right click on the class in the Diagram or Explorer pane. The second item on the speedmenu is Properties. Click it to bring up the properties inspector (in the foreground below). Note the pushpin (⚲) in its lower left corner.



Name	Value
name	CashSaleDetail
package	ProblemDomain
stereotype	mi-detail
alias	
file	CashSaleDetail.java
public	☑
final	☑
abstract	□
extends	
implements	

Press Ctrl+Enter to finish editing and close Inspector

Clicking the pushpin on the Properties Inspector docks it on the bottom of the Explorer pane. To undock, click the pushpin (⚲) in the upper right corner.



Name	Value
name	CashSaleDetail
package	ProblemDomain
stereotype	mi-detail
alias	
file	CashSaleDetail.java

Press Ctrl+Enter to finish editing and close Inspector

EXERCISE

TOP NEXT START HOME

Last Revised: Wed, Apr 11, 2001

Together Quick Tour
Part 2: The Explorer Pane

Together's Explorer pane is a powerful feature for navigation, control, and even code-generation. This part of the tour examines all facets Explorer pane in detail.

You will see how to use the Explorer to get information about the current project and the file system. You will see how to use the Explorer pane to access existing code modules. And you will get a glimpse of how Together is extended via built-in and custom building blocks.

The context for all of our discussions is the sample **CashSales** project.

Contents:

Exploring the Explorer pane

The Explorer pane is organized in tabs.

	Directory	physical structure of the open project and the file system
	Model	logical view of the project's model elements
	Diagram	listing of project diagrams by type
	Overview	thumbnail overview of the Diagram pane
	Components	reusable component models
	Modules	custom building blocks

The Directory and Modules tabs are always present. The Model and Overview tabs are present only when there is an open project. The Diagrams tab and the Components tab are present on request only.

To see the Explorer pane only, bring the Explorer pane into focus, then click the full screen toggle button () on the Main menu.

Directory tab: navigating the file system

The Directory tab shows the system directory structure relative to Together. The display has the following top-level directory nodes.

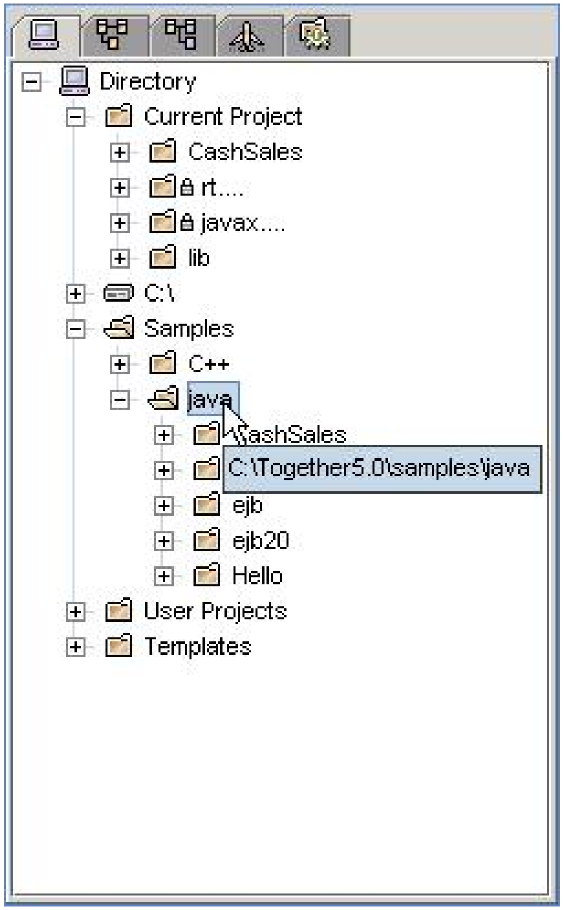
- Current project (if a project is open)
- Top-level physical system directories
- Samples -- directory of sample projects that ship with Together
- User projects -- default Together directory for your personal work
- Templates -- Together templates for C++, Java, and CORBA IDL

When you open a project, a Current Project node appears at the top of the listing. You can expand that node to see the physical files making up the project.

Double-clicking on a project file opens it in Together. Double-clicking on a text file opens it in the Together editor (even if no project is open).

The snapshot here shows a Directory tab with the cursor over the **java** folder. The flyover text is the folder's pathname. (A flyover box appears whenever the cursor is over an actual folder or file.)

The Directory tab allows you to navigate the physical file system and the current project's physical structure.



Note: You can edit the **navigator.config** file to limit which drives the directory tab shows. This is especially useful for networked mapped drives.

EXERCISE

Model tab: examining the logical view of an open project

The Model tab exists when there is an open project. It provides a logical view of the major elements making up the project model.

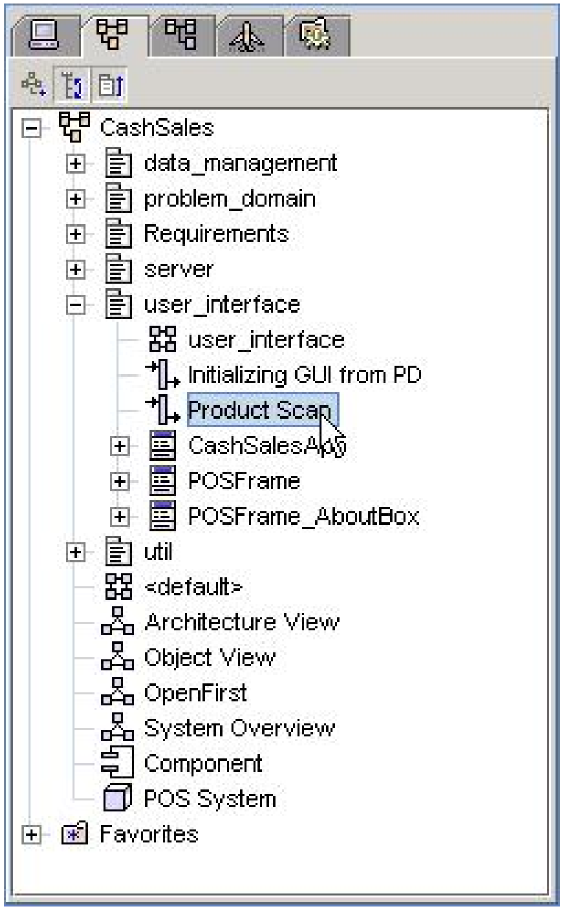
The Model tab shows everything in the project -- packages, diagrams, classes, and interfaces (Java) -- all organized into a tree. The tree root corresponds to the project itself. The second level nodes include:

- Project packages (subpackages are on lower levels). Our snapshot shows several: **data_management**, **problem_domain**, **Requirements**, **server**, **user_interface**, and **util**.
- Diagrams generated from the Main file menu, such as the deployment diagram (POS System) in our snapshot
- **<default>** top-level model
- Top-level interfaces

The Model does not necessarily reflect the physical structure of the project files, since project packages can reside virtually anywhere in the system.

The flyover text in the snapshot is the fully qualified name of the class under the cursor.

Frequently accessed model elements can be stored in the **Favorites** folder at the bottom.



The Model tab has a small toolbar at the top that gives choices for the tree view. The toolbar has three buttons.

- Make diagram nodes expandable to show diagram content (toggled off by default).
- Sort package tree nodes alphabetically (toggled on by default).
- Display packages first, before all other items (toggled on by default).

Diagrams have special icons, such as for a UML sequence diagram. Double-clicking on the diagram in the Model tab opens it in the Diagram pane. You can use the diagram speedmenu to open it in a new Diagram pane tab rather than in the currently open pane. (You can also set the options to have a double click open the diagram or other element in a new tab.)

EXERCISE

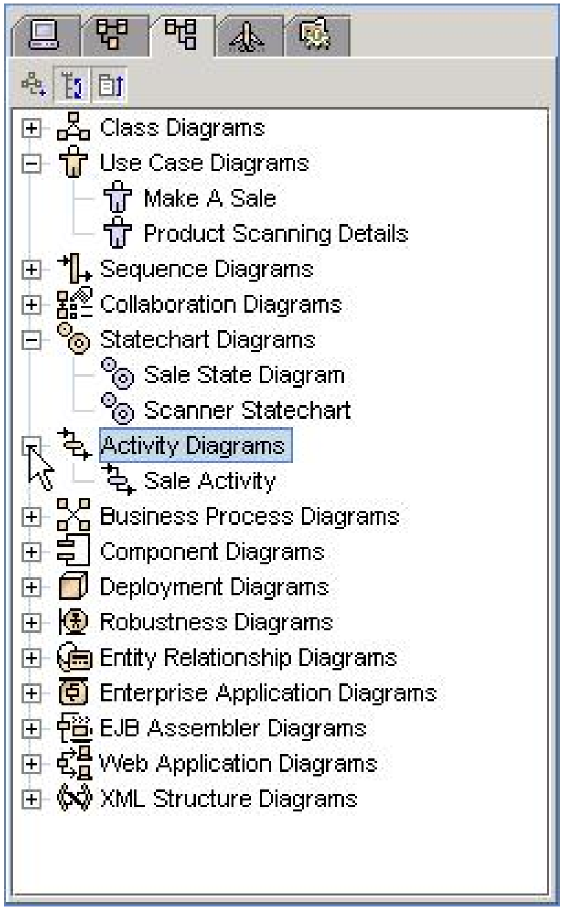
Diagram tab: organizing project diagrams

The Diagram tab gives a listing of all the diagrams in a project, organized according to type. It appears if **Show Diagrams tab** flag is checked on the General page of the Options dialog.

Each node in the Diagram tab corresponds to a diagram type. This tab displays the tree view of all types of diagrams available in Together.

All diagrams in the current project show up in the appropriate nodes. Expand a node in the usual manner by clicking it. Open a diagram in the Diagram pane by double-clicking it in the Explorer.

The snapshot here shows a Diagram tab with the listing of Activity diagrams expanded.



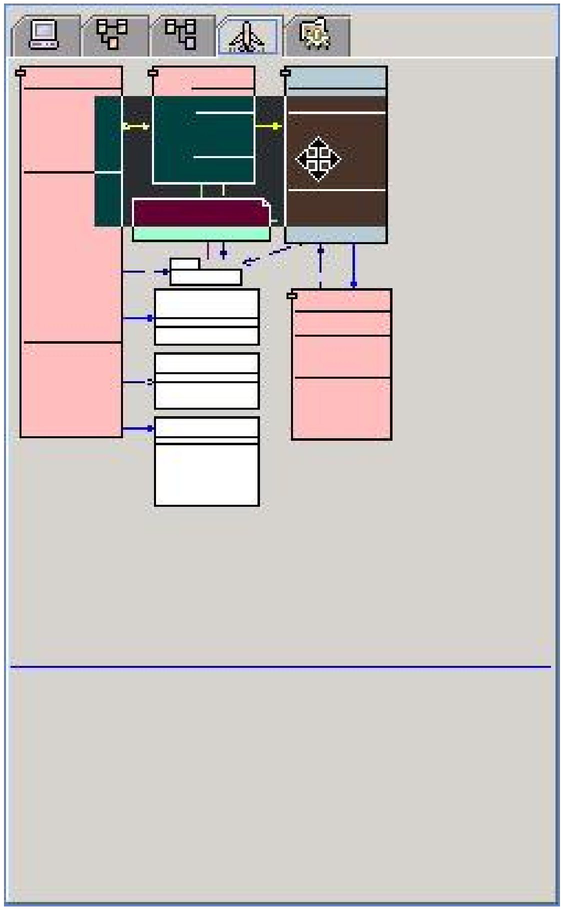
EXERCISE

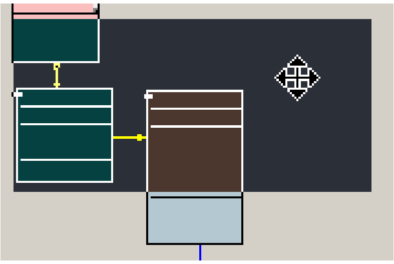
Overview tab: controlling the diagram view

The Overview tab gives a thumbnail sketch of the Diagram pane, placing a shadow over the part of the diagram that is currently visible. You can use the Overview to control the size and location of the visible part of the diagram.

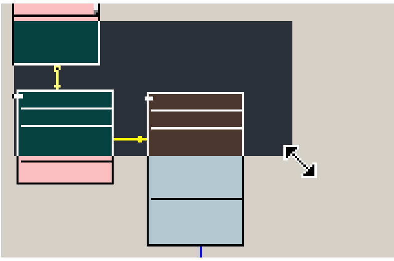
The Overview shadow tightly corresponds to the visible region in the Diagram pane. Any change in that region forces a change in the Overview and vice versa. This includes changes forced by resizing the window or the panes. (The proportions of the shadow are always constrained to match the proportions of the Diagram pane.)

The cursor changes in the Overview as it goes over the shadow. The shape of the cursor indicates move or zoom modes.





To move the visible region without resizing it, hold down the left-mouse button while moving the cursor.



To resize the visible region, grab the lower right corner of the shadow and drag.

EXERCISE

Components tab: accessing and reusing component models

The Components tab allows you to access and reuse component models. It is available only when a project is open. To see Components with the **CashSales** project, invoke the project inspector via the main menu:

File|Project Properties...

Check "Include Components" on.

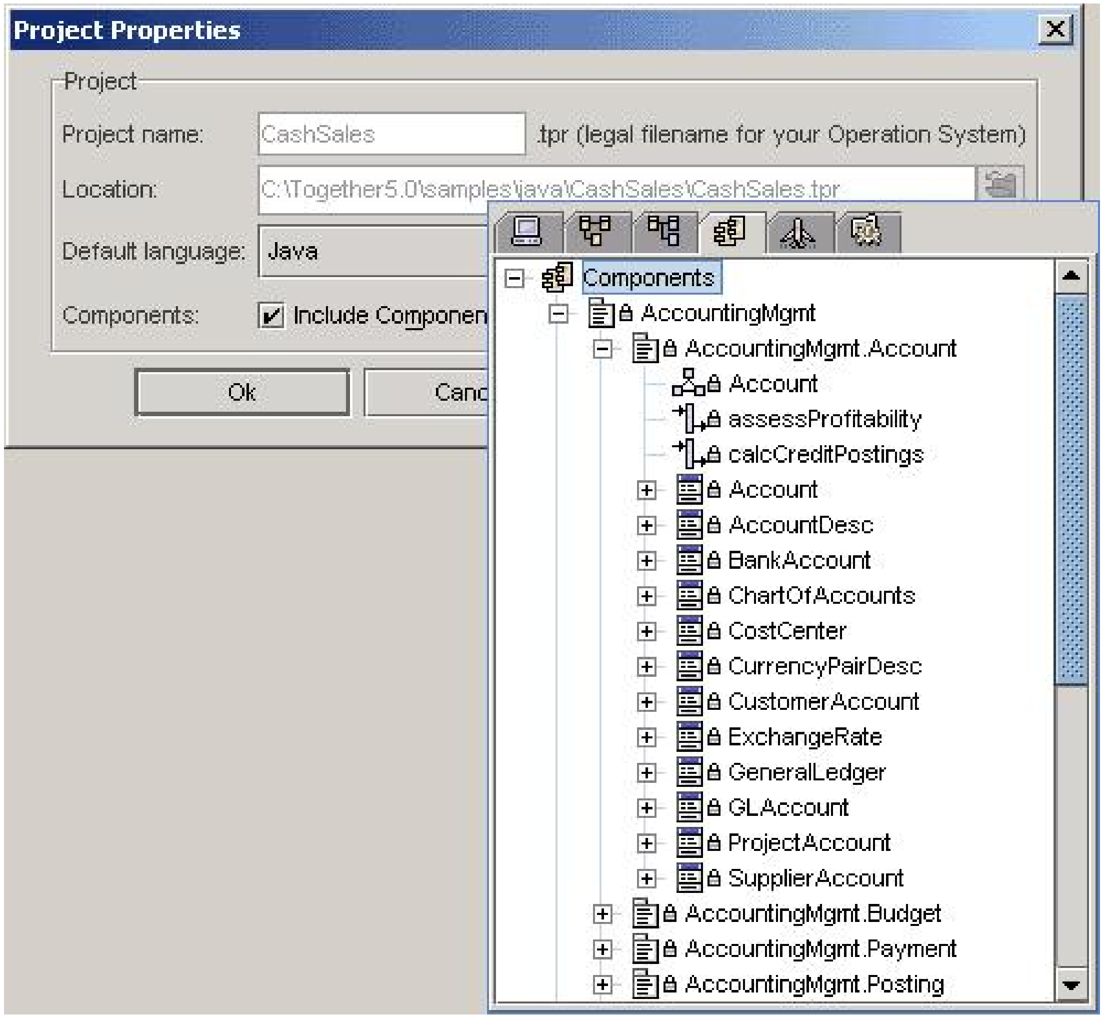
The **CashSales** project uses Coad Modeling Components. These enterprise component color models are part of the Together installation.

All components reside in Together's **modules/components** directory.

You can copy packages and classes shown in the Explorer Components to your class diagrams or to any package in your project.

You can also create your own components. Place them in **component** subdirectories for use in projects.

Our snapshot shows some of the Coad Modeling Components. The small lock on an icon indicates read-only.



Project Properties

Project

Project name: CashSales .tpr (legal filename for your Operation System)

Location: C:\Together5.0\samples\java\CashSales\CashSales.tpr

Default language: Java

Components: ☑ Include Componen

EXERCISE

Modules tab: accessing and extending Together building blocks

Together is highly extensible by means of an open Java API. The open API enables you to write Java programs that use model information from Together and interact with Together itself to extend its native capabilities. Such building-block programs are called **modules**.

Modules are stored in subdirectories under:

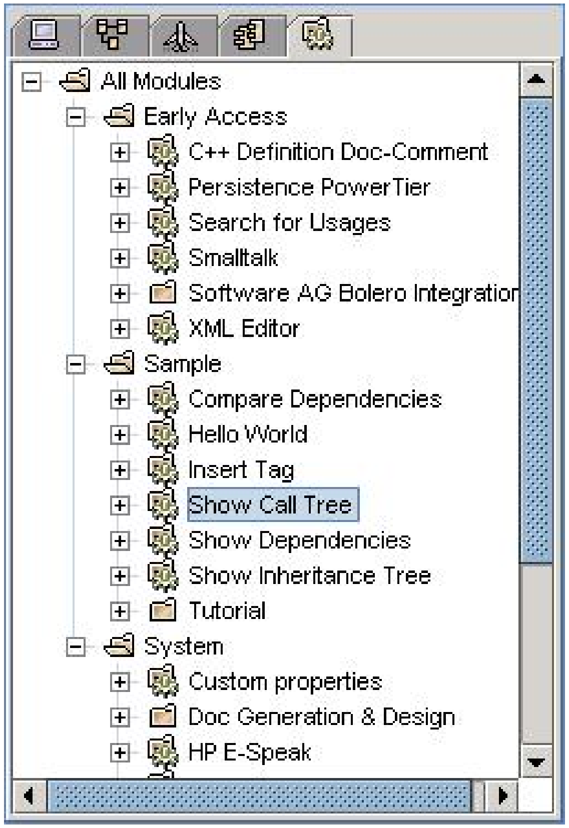
$TOGETHER_HOME$/modules/com/togethersoft/modules

The Module tab gives access to the modules that are supplied by Together, your own modules, and third party modules.

The Explorer Modules tab uses these special icons.

- Module folder. A folder has the same speedmenu as its contents (to run or activate the module).
- Java source code module
- Compiled Java module
- Tcl script module (included for backward compatibility)

The **Sample/Tutorial** directory contains some simple modules that you can examine in the Editor and run. The standard output goes to the Message pane. Use these tutorial modules to discover how to write your own modules.



All Modules
Early Access
C++ Definition Doc-Comment
Persistence PowerTier
Search for Usages
Smalltalk
Software AG Bolero Integration
XML Editor
Sample
Compare Dependencies
Hello World
Insert Tag
Show Call Tree
Show Dependencies
Show Inheritance Tree
Tutorial
System
Custom properties
Doc Generation & Design
HP E-Speak

EXERCISE

← PREVIOUS | TOP ↑ | NEXT → | START HOME

Last Revised: Thu, Apr 12, 2001

Together Quick Tour
Part 3: Navigating Within Together

Together is a powerful architectural tool that maintains consistent presentation of the project at all times. The Explorer pane is always in sync with the actual project. And the code in the Editor pane is always in sync with the model class diagrams.

In this part of the tour, you will look at the connections between the Explorer pane and the Diagram and Editor panes. And you will examine the project presentation in the Diagram and Editor panes to see how closely knit the two views are.

The context for all of our discussions is the sample **CashSales** project.

Contents:

Opening and closing diagrams from the Explorer pane

Every project has a top-level default diagram. Unless you change its properties, the first time you open a project, the default diagram opens in the Diagram pane.

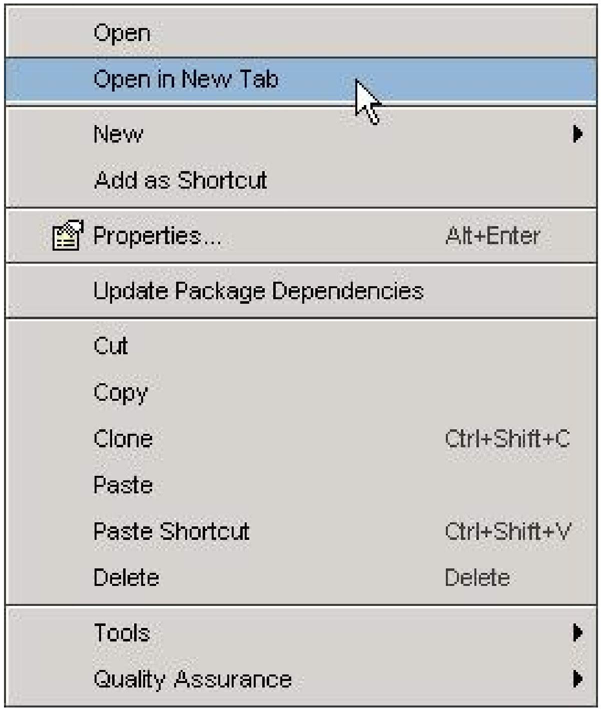
Diagrams can be opened in the Diagram pane only from within their projects.

You can open a diagram from the Main menu (under File). But it is easier to open a diagram from the Model tab (⊞) in the Explorer pane by using one of these two methods.

- Double click on the diagram (name or icon). It will open in the Diagram pane, closing the current focus diagram (if any).

  --or--

- Right click on the diagram in the Model tab to get its speedmenu, then select **Open in New Tab**. This opens the diagram in a new tab without closing the current one.



There are different types of diagrams, each with an associated icon. ⊞ is the default diagram icon. Together's UML diagram icons are abstractions of actual diagrams.

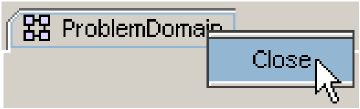
class/object	sequence	statechart	component
use case	collaboration	activity	deployment

The Model tab of the Explorer displays diagram files with those icons. The tabs of open diagrams in the Diagram pane uses the icons as well.

Together also has seven special diagram types, mostly for Enterprise or J2EE modeling.

To close a diagram without closing the project, right-click its Diagram pane tab. This brings up a clickable close box like that on the right.

You can also close a diagram from its Diagram pane speedmenu. Right click on an empty region of the diagram to get its speedmenu.



EXERCISE

Opening source code files from the Explorer

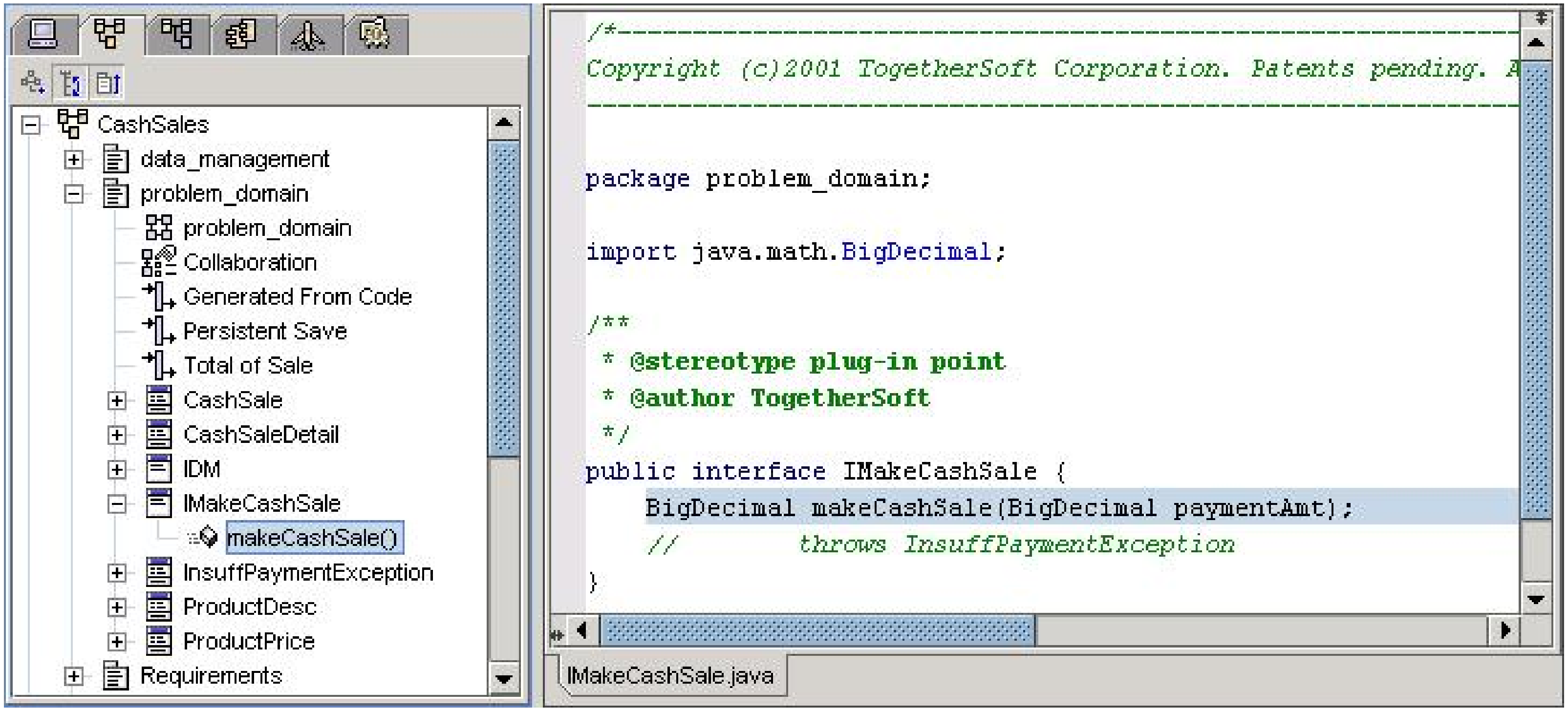
You can open files from three Explorer tabs: the Directory tab ( ), the Model tab ( ), and the Diagram tab ( ).

In the Directory tab, you can open project files ( ) and text files ( ). Double-clicking on a text file opens it in the Editor pane. Conversely, when you create a new file (source code or diagram), it will show up in the Explorer.

The Model tab uses three special file icons:

class source file	interface source file	package directory

Double-clicking on a class or interface in the Model tab opens its source code file in the Editor pane. The snapshot below shows using the model tab of the Explorer pane to jump to a method in the Editor.



```
/*-----------------------------------------------------------------
Copyright (c)2001 TogetherSoft Corporation. Patents pending. A
-------------------------------------------------------------------

package problem_domain;

import java.math.BigDecimal;

/**
 * @stereotype plug-in point
 * @author TogetherSoft
 */
public interface IMakeCashSale {
    BigDecimal makeCashSale(BigDecimal paymentAmt);
    //        throws InsuffPaymentException
}
```

Double-clicking the diagram name in the Diagram tab opens it in the Diagram pane.

You can also open a file from the Editor pane speedmenu. Choose **Open** to get a file selection box. Unlike the Explorer, the Editor allows you to open any type of file.

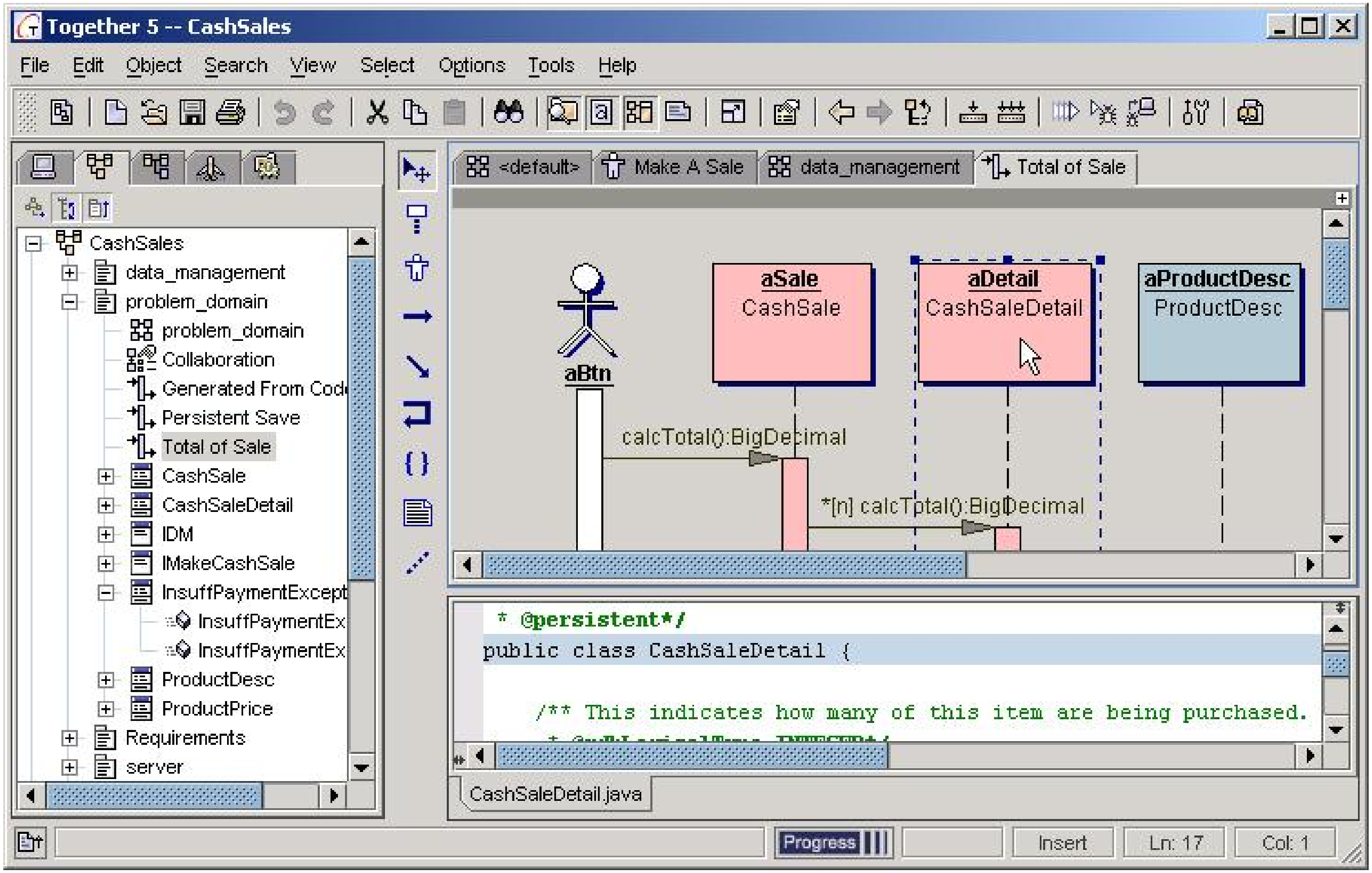
EXERCISE

Integrating views -- Explorer to Diagram to Editor

Sequence diagrams and class diagrams are related directly to source code. We'll use the **CashSales** project to illustrate some of these connections.

To get the Main window below, follow through these steps:

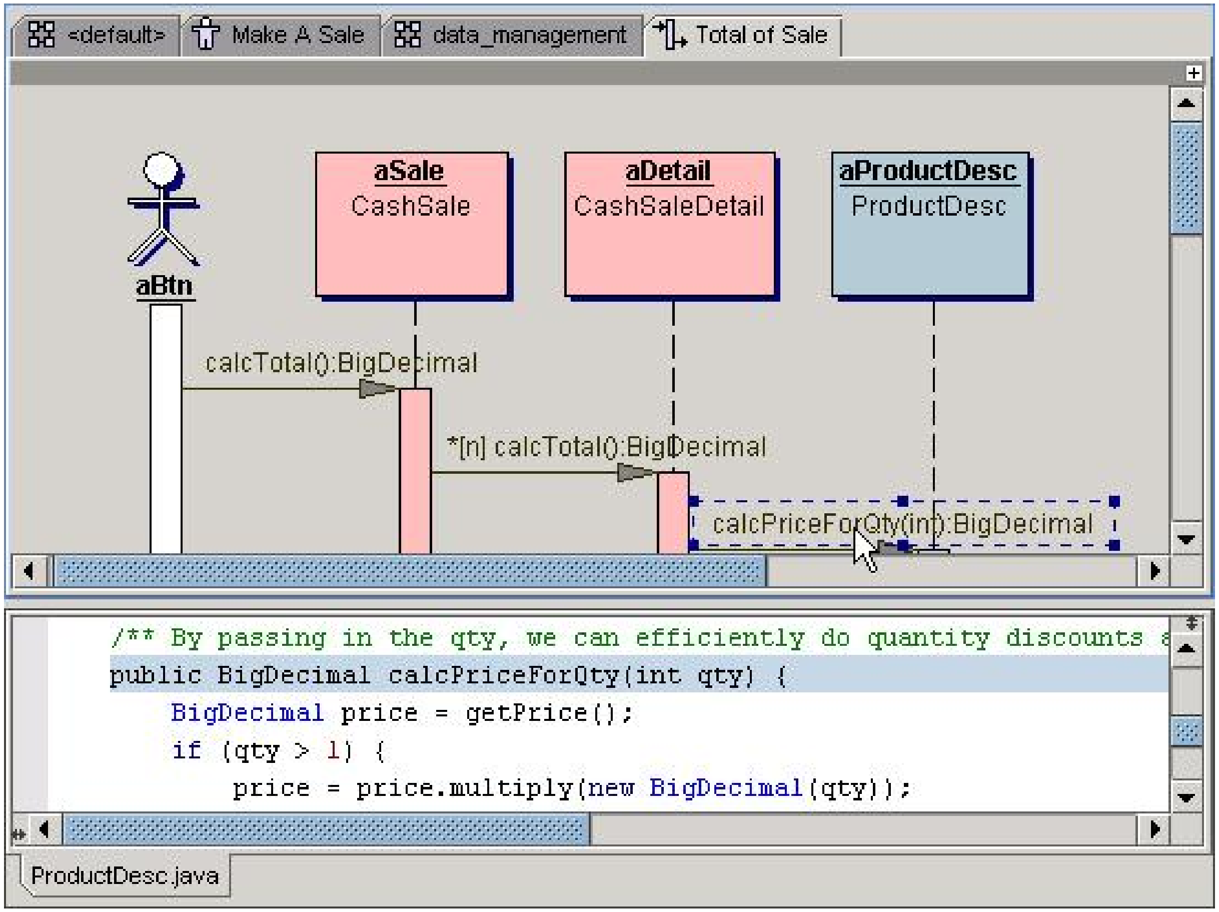
1. Expand the **problem_domain** package in the Model tab of the Explorer pane.
2. Open **Total of Sale** sequence diagram from the Model pane.
3. If needed, adjust the sliding bars on the Diagram pane to scroll to **aDetail**. (Alternatively, use the overview tab of the Explorer to position the viewing region.)
4. Select the **aDetail** object in the Diagram pane by clicking on it.



```
 * @persistent*/
public class CashSaleDetail {

    /** This indicates how many of this item are being purchased.
```

When you select an object in a sequence diagram, the Editor automatically scrolls to the corresponding class code and highlights the first line of its definition.

The snapshot to the right shows selecting the call to **calcPriceForQty()** on the sequence diagram. The method becomes highlighted in the Editor.



```
/** By passing in the qty, we can efficiently do quantity discounts a
public BigDecimal calcPriceForQty(int qty) {
    BigDecimal price = getPrice();
    if (qty > 1) {
        price = price.multiply(new BigDecimal(qty));
```

EXERCISE

← PREVIOUS | TOP ↑ | NEXT → | START HOME

Last Revised: Thu, Apr 12, 2001

Together Quick Tour
Part 4: Customization via Options and Properties Inspectors

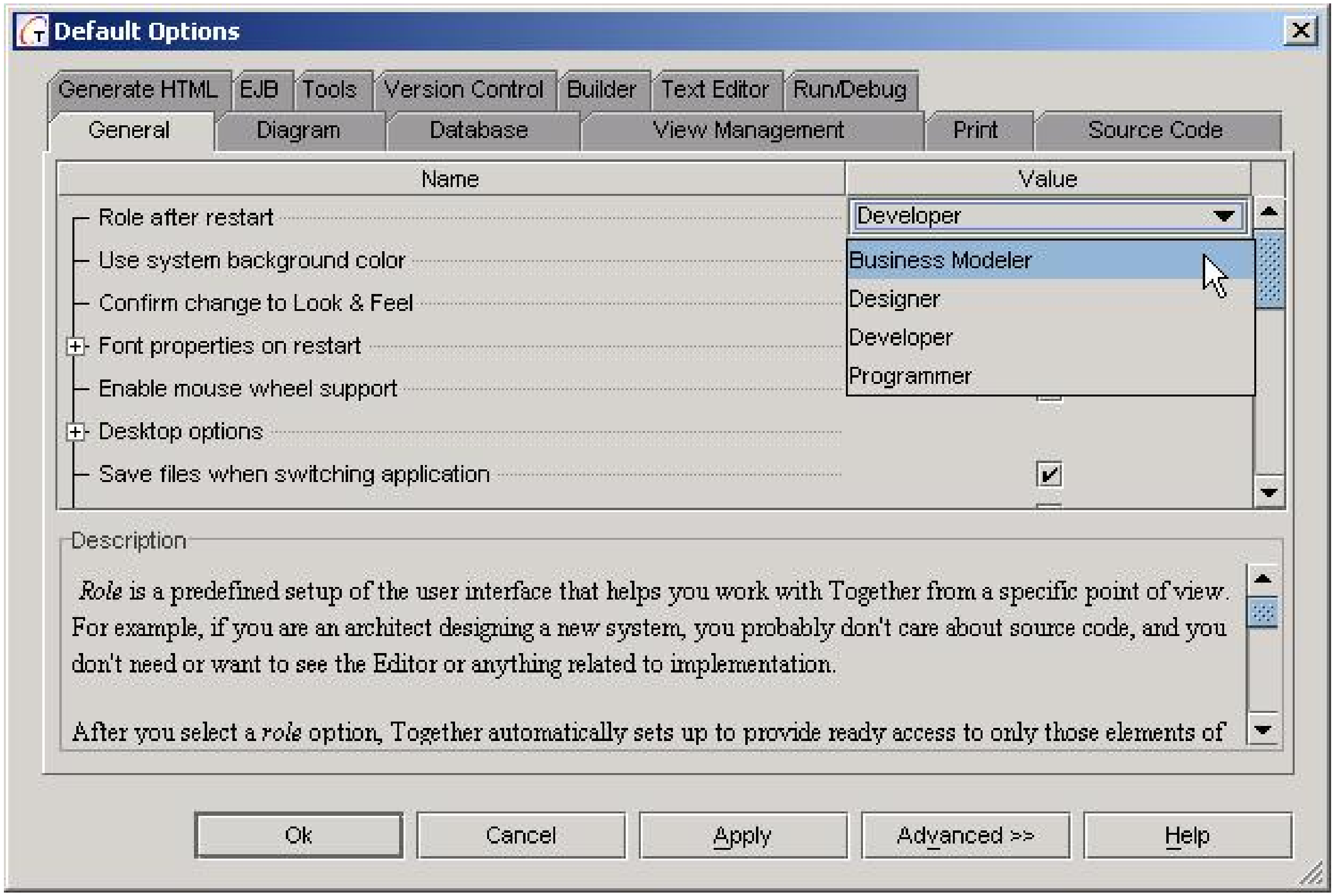
Together gives users many ways to customize their workspaces (including changing the text configuration files). In this part of the Together Quick Tour, you will learn how to use the properties inspectors and options menus to do the customization. When you are finished, you're ready to begin the Together Tutorial.

The context for all of our discussions is the **CashSales** project.

Contents:

Accessing default and project options

Default, project, and diagram options are available via the **Options** command on the Main menu. The dialog box shown below comes from the **Options|Default** command. Options are organized by tabs; the General tab is at the front in this snapshot.



Name	Value
Role after restart	Developer
Use system background color	
Confirm change to Look & Feel	
Font properties on restart	
Enable mouse wheel support	
Desktop options	
Save files when switching application	☑

Description

Role is a predefined setup of the user interface that helps you work with Together from a specific point of view. For example, if you are an architect designing a new system, you probably don't care about source code, and you don't need or want to see the Editor or anything related to implementation.

After you select a *role* option, Together automatically sets up to provide ready access to only those elements of

Some settings have check boxes. Some are on pulldown menus. Some have text boxes that can be edited in place. Clicking on a setting name brings up its description on the lower part of the pane.

The snapshot above shows configuring Together for different workspace roles. For example, the Business Modeler role focuses on not exposing extra details in Diagram pane. The Editor pane shows only on demand, and menus and toolbars are simplified.

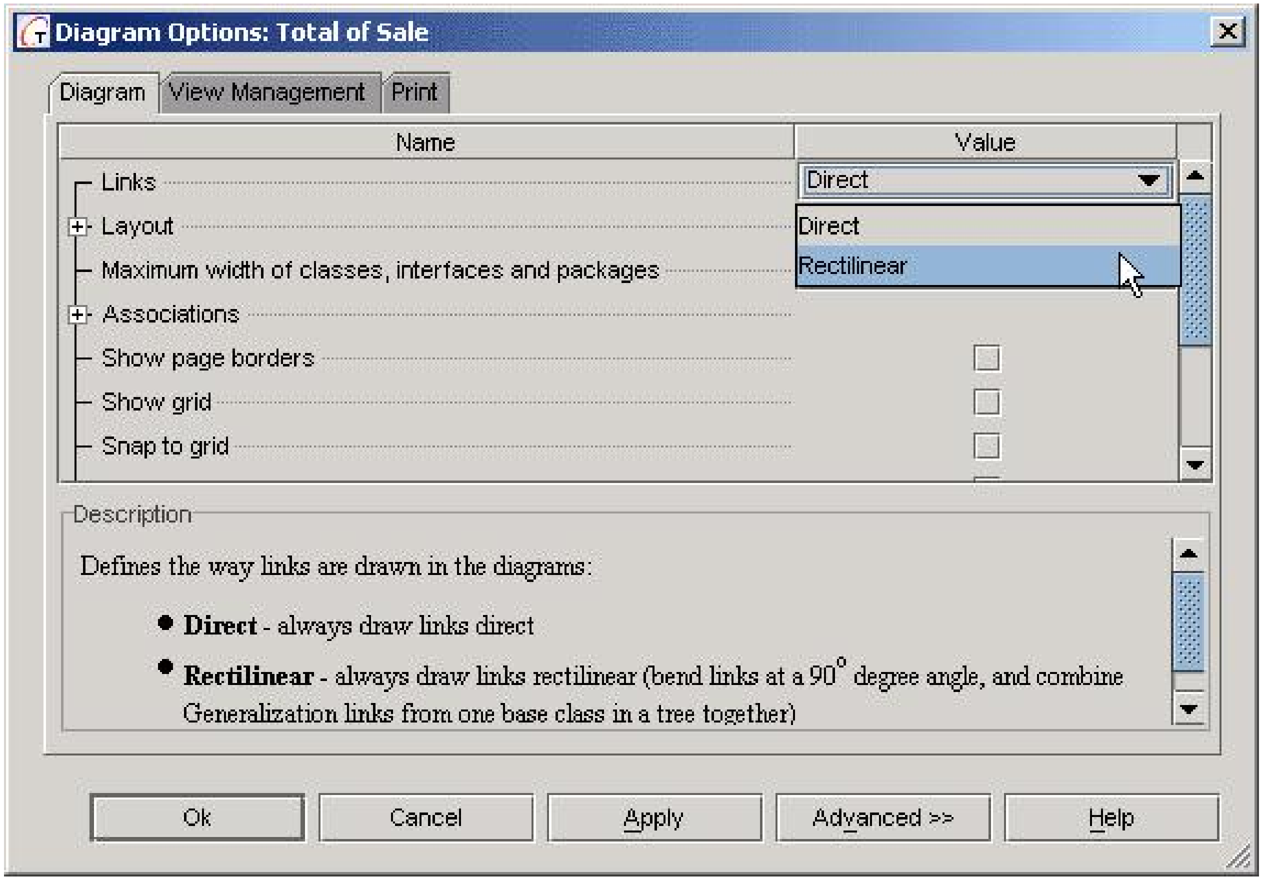
The default workspace role is Developer. Changes in the workspace role setting take place when you restart Together, when you select **Options|Reload** from the Main menu, or when you click **Apply** on the options dialog box.

Many of the project options (**Options|Project**) are available as default options. They can be applied at a project level or as defaults for all projects.

EXERCISE

Changing diagram options

Diagram options control the diagram presentation views. You can access a diagrams options by selecting **Diagram Options** on its speedmenu.



Name	Value
Links	Direct
Layout	
Maximum width of classes, interfaces and packages	
Associations	
Show page borders	□
Show grid	□
Snap to grid	□

Defines the way links are drawn in the diagrams:

- **Direct** - always draw links direct
- **Rectilinear** - always draw links rectilinear (bend links at a 90° degree angle, and combine Generalization links from one base class in a tree together)

You can apply option settings to a particular diagram or to an entire project. Or you can apply them as defaults for diagrams in all projects.

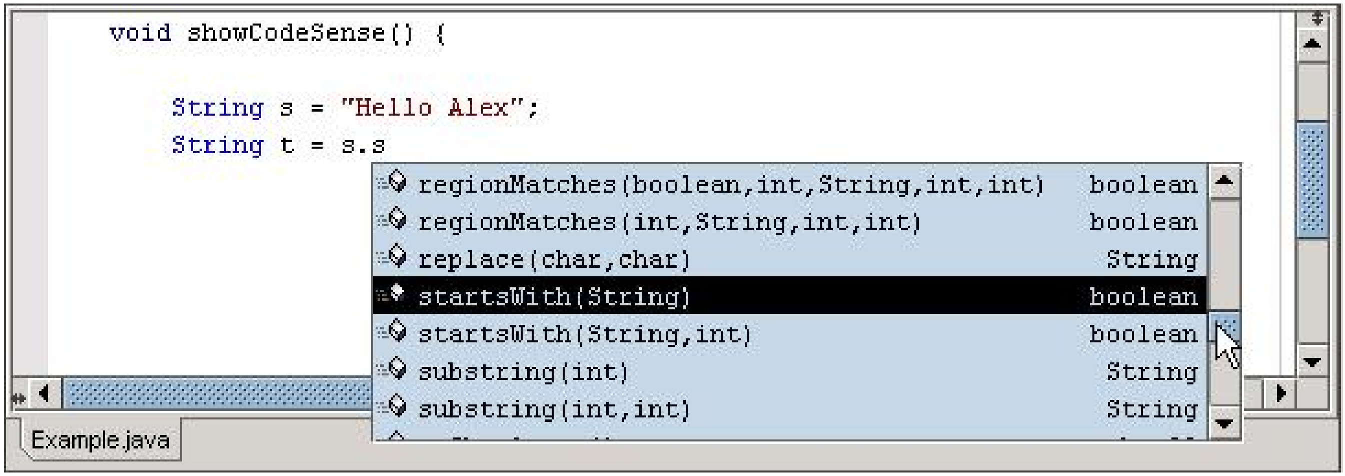
EXERCISE

Changing editor options

Look for the **Text Editor Options** on the Editor speedmenu. There are several choices for settings.

- Font size and cursor orientation
- CodeSense (automatic code completions for Java statements)
- Keyboard hot keys
- Schemes for the project language, such as keyword color, etc.
- External editors, what they are and Together menus on which they'll appear

CodeSense works with the Java libraries to complete statements. The snapshot below comes from an Editor with activated CodeSense. To get the snapshot below, we started inside the body of a method. Then we typed **s.s** to pop up a menu of available **String** methods.



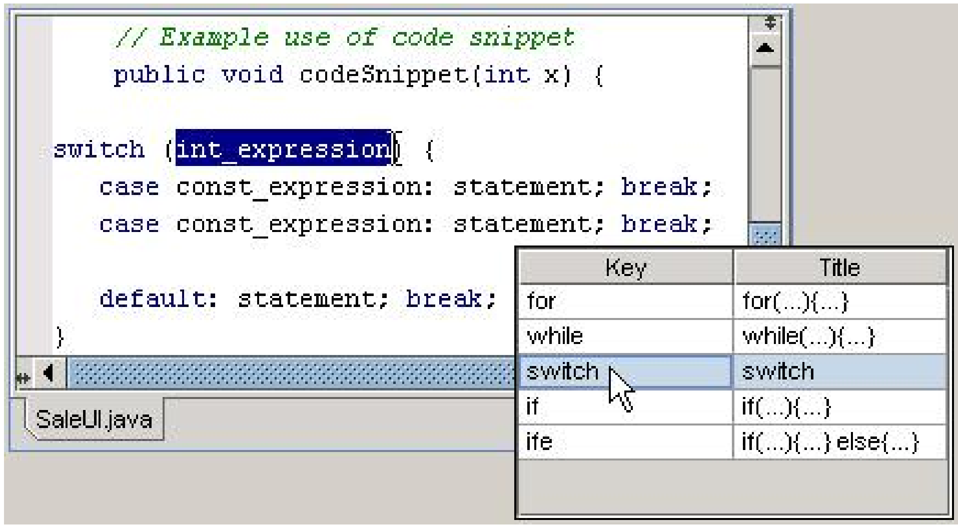
```
void showCodeSense() {

    String s = "Hello Alex";
    String t = s.s
```

```
regionMatches(boolean,int,String,int,int)   boolean
regionMatches(int,String,int,int)           boolean
replace(char,char)                           String
startsWith(String)                          boolean
startsWith(String,int)                      boolean
substring(int)                               String
substring(int,int)                           String
```

The Together Editor recognizes source code files in the language of the current project. With Editor Schemes you can tailor the Editor for your project language, including such things as keyword color and auto indent. You can also build templates for commonly used code constructs.

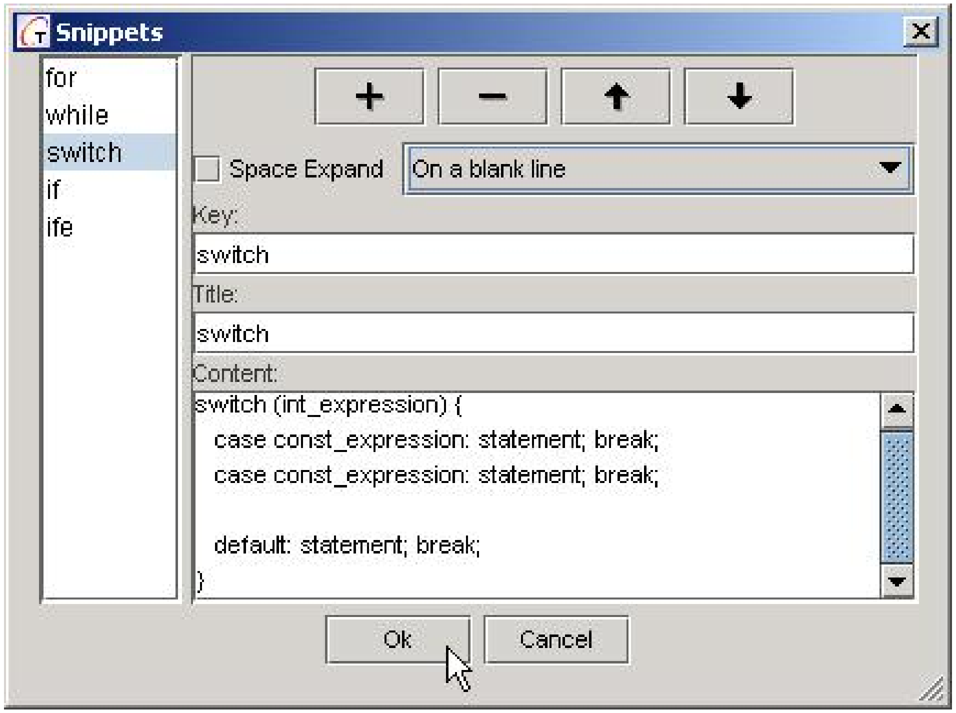
The snapshot to the right shows how we created a new Java **switch** statement template. In the Editor options, select **Schemes|Java|Snippets** to get the Snippets window.

Back in the Editor, we were able to place the template in the code by typing the snippet name (**switch**) then Ctrl+J.



```
    // Example use of code snippet
    public void codeSnippet(int x) {

switch (int_expression) {
   case const_expression: statement; break;
   case const_expression: statement; break;

   default: statement; break;
}
```

Key	Title
for	for(...){...}
while	while(...){...}
switch	switch
if	if(...){...}
ife	if(...){...} else{...}



□ Space Expand On a blank line

Key: switch

Title: switch

Content:

```
switch (int_expression) {
   case const_expression: statement; break;
   case const_expression: statement; break;

   default: statement; break;
}
```

EXERCISE

Customizing objects through properties inspectors

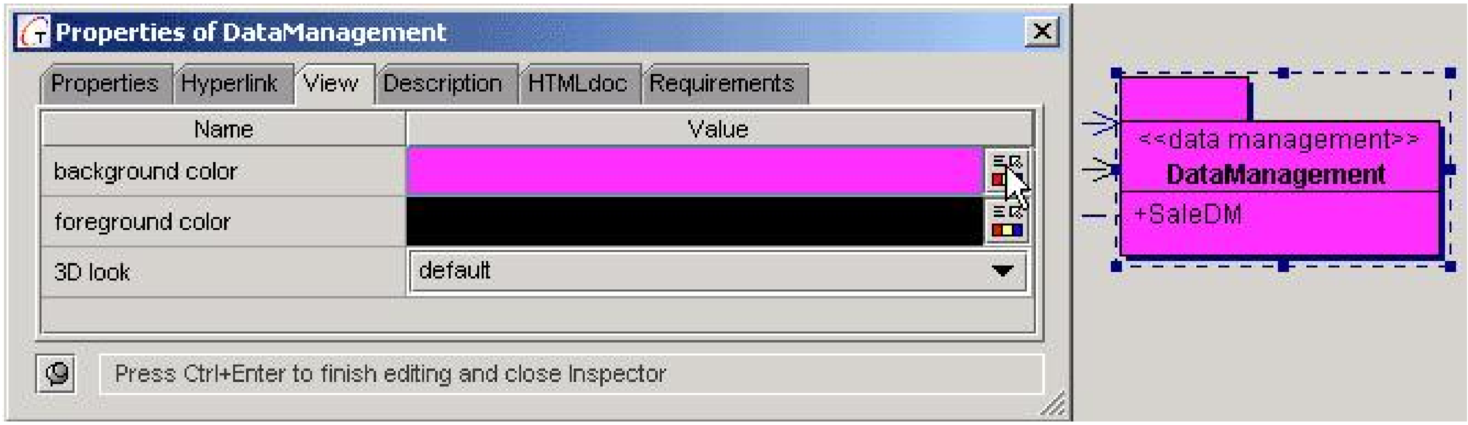
Together considers many elements to be objects, including:

- diagrams
- most diagram elements
- classes
- java interfaces
- packages

Each Together object has a speedmenu with a **Properties** command, which brings up the object's properties inspector. The properties inspector lets you customize the object's look, behavior, characteristics, and documentation.

Properties inspectors vary according to the kind of object. The picture here illustrates using a properties inspector to customize the color of the **data_management** package.

We weren't very pleased with that fuchsia. So we undid setting the color scheme by clicking the undo button (↶) on the Main toolbar.



Properties of DataManagement

Name	Value
background color	🟪
foreground color	⬛
3D look	default

Press Ctrl+Enter to finish editing and close Inspector

EXERCISE

← PREVIOUS | TOP ↑ | START HOME

Last Revised: Thu, Apr 12, 2001

Together Tutorial
Part 1: Projects and Packages

The scope of the Together Tutorial includes the most commonly used features of Together. We'll show you how to construct your own project from the beginning. If you are brand new to Together and think you might have trouble navigating your way around, go through the Together Quick Tour before beginning the work here.

One of the early and continuing hallmarks of Together is its ability to keep class model and code in sync -- all the time, every time. It's what Together calls LiveSource^{TM} technology, and you'll get a first peek at it here.

Contents

Creating a new project from scratch

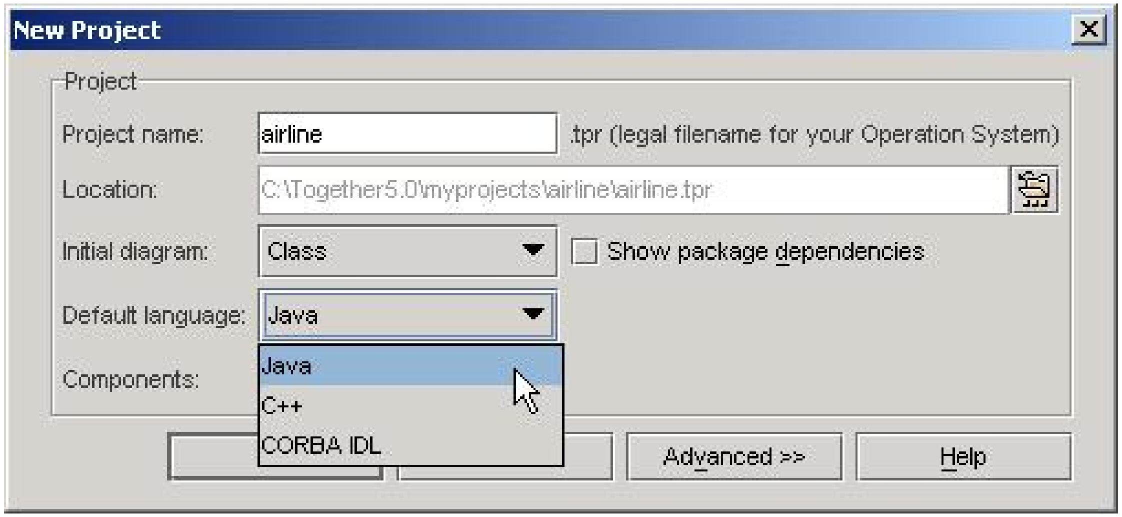
Most of the Together Tutorial is centered around this sample problem.

> "A small regional airline needs an application for keeping track of flight reservations and ticket revenues."

The first step in tackling this problem is to set up a Together project for developing a problem solution.

Step: Create a new Java project named **airline**.

Open Together and select **File|New Project** from the Main menu. You'll see a New Project dialog box like the one below.



New Project

Project

Project name: airline .tpr (legal filename for your Operation System)

Location: C:\Together5.0\myprojects\airline\airline.tpr

Initial diagram: Class

□ Show package dependencies

Default language: Java

Components:

Unless you specify otherwise, Together creates a new directory for the new project inside **$TOGETHER_HOME$/myprojects**. The name of the directory is the same as the project name. Together gives a choice of three languages: Java, C++, or CORBA IDL.

At a minimum, a project consists of:

- a project file (with the **.tpr** file extension and icon)
- a default package diagram (**default.dfPackage**)
- a primary root directory

When we created **airline**, Together created three files in the primary root directory **airline**.

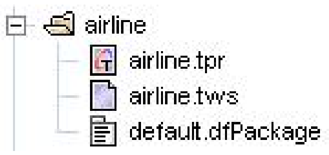
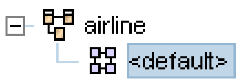
- **default.dfPackage**
- **airline.tpr**
- **airline.tws** (Together workspace settings)

The User Projects folder in the Directory tab of the Explorer pane corresponds to the physical directory **$TOGETHER_HOME$/myprojects**. The primary root directory of **airline** is under User Projects. And since **airline** is open, its primary root directory appears also under Current Project.

Working with the <default> diagram and primary root directory

Together generates a <default> diagram for each new project. The <default> diagram shows packages of the primary root directory as well as classes of any source code files in that directory. (Default diagrams and diagrams for packages have the icon ▦. They are class diagrams.)

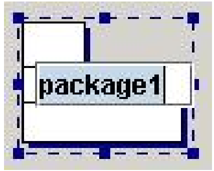
When a project is first created, the <default> diagram is simply a blank background. Below are two views of the newly created project in the Explorer pane.

 airline airline.tpr airline.tws default.dfPackage	Directory tab view. The primary root directory (**airline**) contains no other directories when the project is first created. Together uses the suffix **.tpr** to indicate that this is a project file. The suffix **.tws** is for Together workspace settings. All three files are ASCII files.
 airline <default>	Model tab view. There is only one part of the model. It contains no elements at the start.

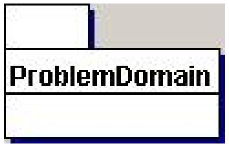
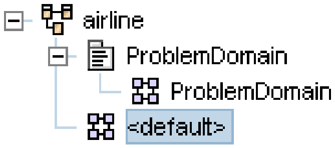
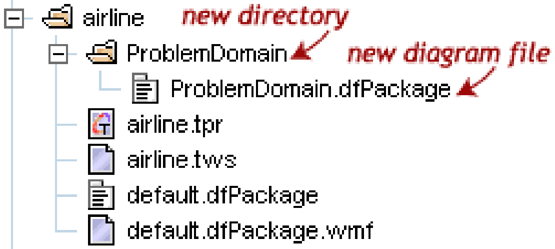
Creating new packages

The <default> diagram is the place to start organizing a project into packages. The **airline** project will have three packages.

Step: Create a new package named **ProblemDomain** inside the <default> diagram.

To create a new package, click on the package button (▤) of the vertical Diagram toolbar. Then click on the diagram background. The diagram will get a new node. At this point, you can edit the package name by typing directly in the in-place editor that is now active. Press Enter to apply the name.	 package1

As you make a package, you can see the Together's LiveSource technology go to work -- Together automatically creates a physical directory for the package and generates a default diagram inside the directory. The new diagram will show any physical project content Together finds now or later.

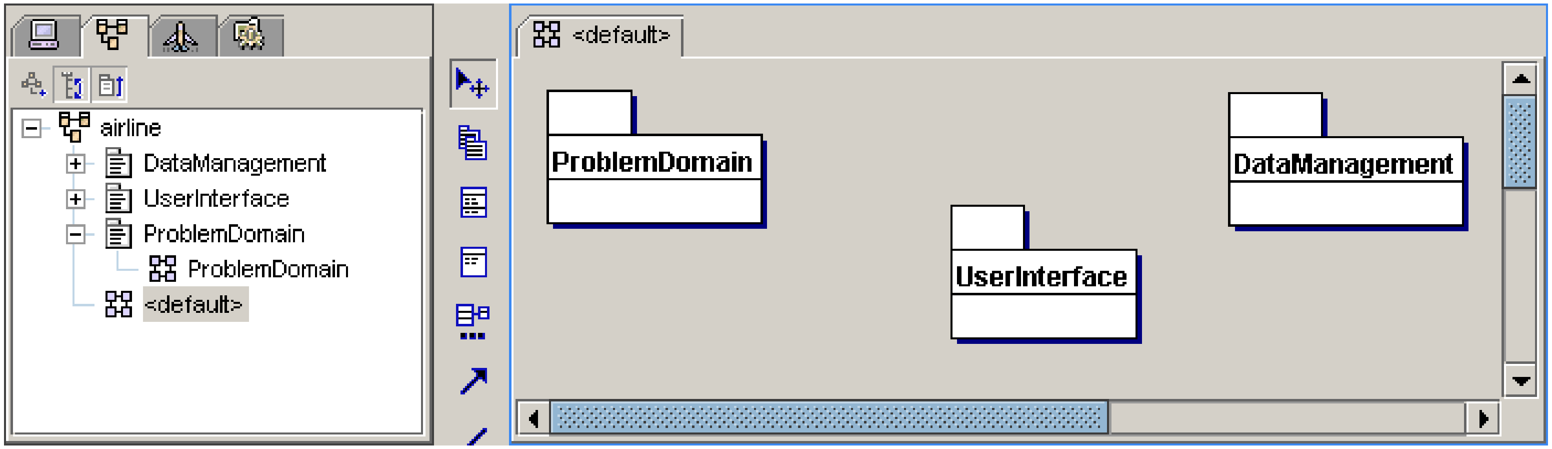
In the Diagram pane, the <default> diagram now contains a single node, which is a package.	 ProblemDomain
The Model tab of the Explorer pane shows the new Package node. Inside the new package is a node for another diagram, which has the same name as the package. Both the new package and the new diagram are currently empty.	 airline ProblemDomain ProblemDomain <default>
The Directory tab of the Explorer shows the new file structure of the project. There's a new subdirectory of the primary root directory named **ProblemDomain**. That directory now contains the file **ProblemDomain.dfPackage**, which is the default diagram for the new package. (The **.wmf** file is a Windows metafile.)	 airline *new directory* ProblemDomain *new diagram file* ProblemDomain.dfPackage airline.tpr airline.tws default.dfPackage default.dfPackage.wmf

To see the contents of the new **ProblemDomain** diagram, double click the diagram in the Model tab of the Explorer. Alternatively, use the diagram speedmenu to open the diagram in a new tab.

Step: Create two additional packages in the <default> diagram: **UserInterface** and **DataManagement**.

There's a shortcut for creating multiple packages. Ctrl+Click the package button on the toolbar to keep the button depressed. While the button is depressed, you can click on the Diagram pane, creating packages and renaming them in place without returning to the toolbar for each package.

To release the button, click it again. (If you drop an extra package on the diagram by mistake, the undo button on the Main toolbar will remove it.)



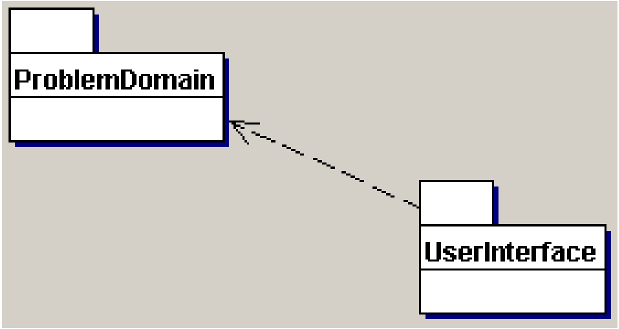
Showing package dependencies

Step: Create a dependency from **UserInterface** to **ProblemDomain**.

The Diagram pane toolbar provides an entire suite of tools for creating UML model elements.

To create a package dependency, click the dependency button (with the dashed arrow icon, (⤢). Then click the dependent package (the "client") in the diagram and drag the end of the arrow to the package that it depends on (the "supplier").

Your dependency should look like the snapshot here.



You can use the dependency's inspector to alter its properties, including changing its label, picking a stereotype, and setting the roles of supplier and client. To get the inspector, right-click on the dependency to bring up its speedmenu, and select Properties.

Tips and Tricks

- It is best to name projects with legal filenames for your operating system. Avoid embedding blanks punctuation marks, or any special characters.
- The suffix **.tpr** indicates Together project. Together keeps all of its files in flat ASCII text -- you can view them with an ordinary editor.

- Always organize your projects in packages.
- To create multiple nodes of the same type, Ctrl+click the toolbar button to keep it depressed. You can even use the in-place editor as you go along. Clicking the button again releases it.

TOP ↑ | NEXT → | START HOME

Last Revised: Thu, Apr 12, 2001

Together Tutorial
Part 2: Requirements and Use Case Diagrams

The first steps of any software project involve nailing down some of its features. The airline problem is potentially huge, but we will concentrate on a small number of requirements.

- Make a reservation.
- Buy a ticket.
- Determine if a flight has room for more reservations.
- Find the total ticket revenue for a flight.

In this part of the Together Tutorial, you will create a Use Case diagram to capture these requirements.

Contents:

Creating a new diagram

Open the **airline** project that you created for the previous part. Bring the **ProblemDomain** diagram into focus (bring it to the front in the Diagram pane).

Step: Create a new Use Case diagram named **MakeReservation**.

Clicking the New Diagram button (▣) on the Main toolbar brings up a New Diagram dialog box.

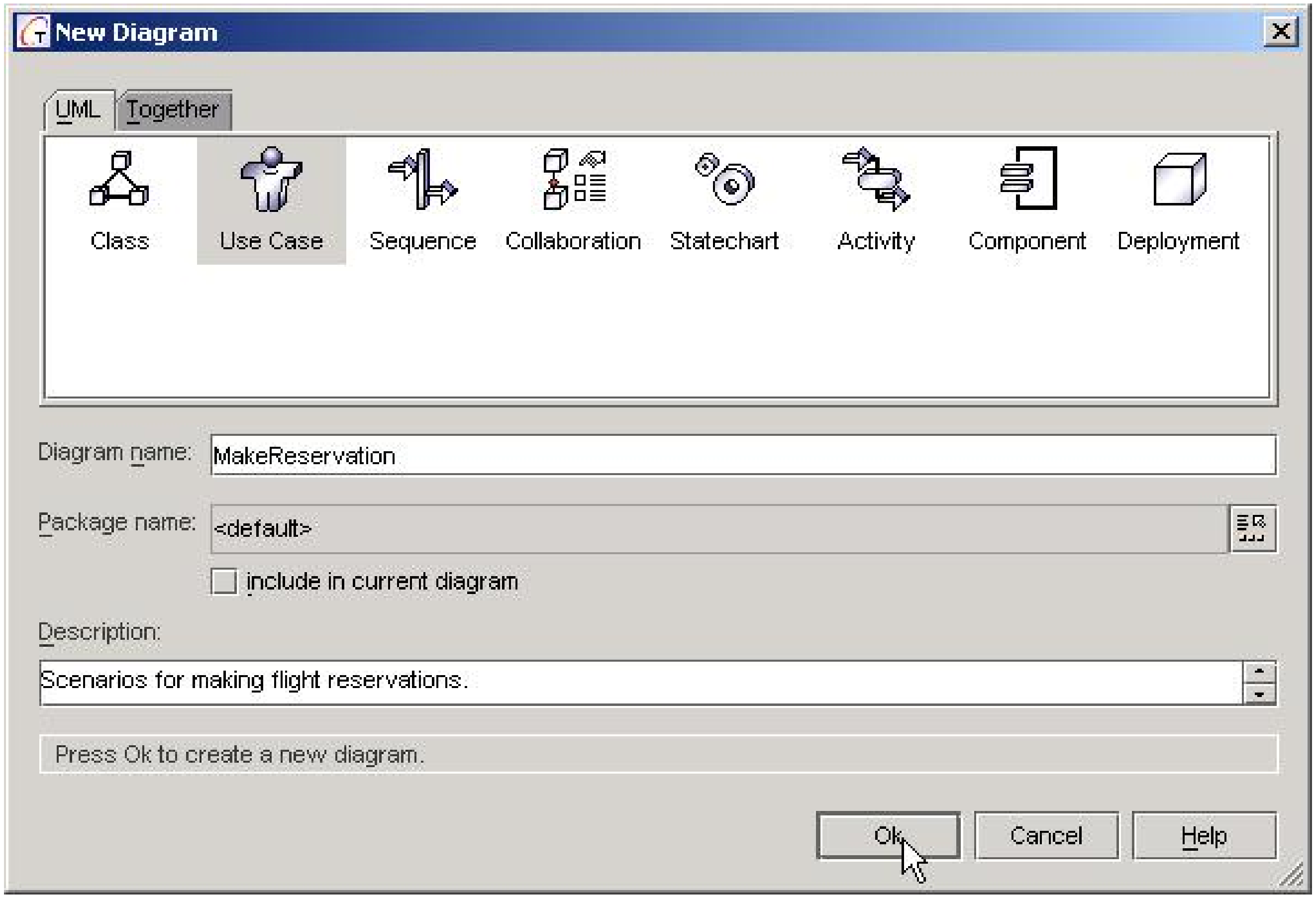


Diagram name: MakeReservation

Package name: <default>

□ include in current diagram

Description:

Scenarios for making flight reservations.

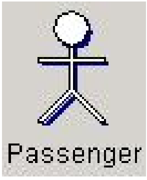
Press Ok to create a new diagram.

When you click OK, the new diagram shows up in the Model and Directory tabs of the Explorer pane.

Creating actors, use cases, and a system boundary

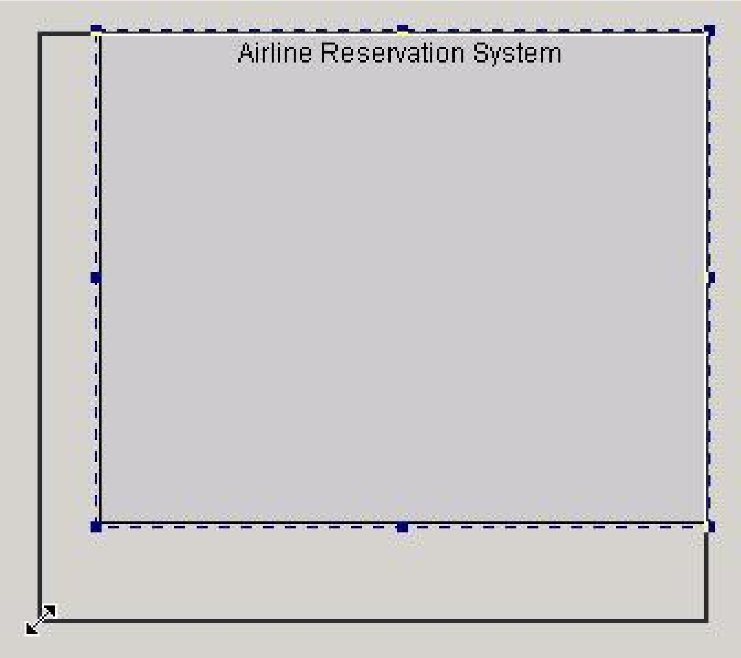
The **MakeReservation** diagram should now be in focus now in the Diagram pane.

Step: Create three actors, **Passenger**, **FinanceOfficer**, and **Agent**.

The Diagram pane toolbar varies according to the type of diagram. Click the actor button (웃) and then the diagram to create a new actor. With the in-place editor, fill in the actor's name.	

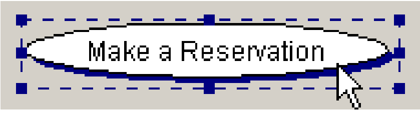
If you Ctrl+click the actor button to place an actor in the diagram, you'll be able to create another actor without first returning to the Diagram toolbar. You can even edit its name as you go along. That's how most of the element buttons on the Diagram toolbars work. To click on the diagram without putting on another actor, simply click the actor button once again.

Step: Put a system boundary on the diagram and name it **Airline Reservation System**.

Click the system boundary button (□) and then the diagram to create the system boundary. Fill in its name with the in-place editor. When you finish, you can move the system boundary and resize. Our snapshot shows resizing by grabbing a corner.	

Step: Create four new use cases:

1. **Make a Reservation**
2. **Check Availability**
3. **Buy a Ticket**
4. **Find Ticket Revenue**

Click the use case button (○) and then on the diagram to create a new use case. Use the in-place editor to fill in a text description.	

You can create a use case then drag it in the system boundary. Or you can create the use case within the system boundary at the start. You can always reposition any diagram element by dragging it with the mouse. A use case within a system boundary will move when the system boundary is repositioned.

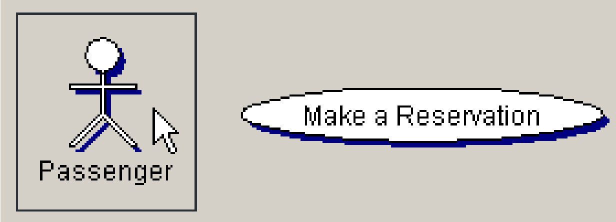
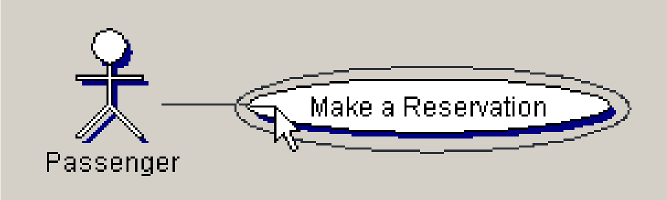
Connecting diagram elements

Communication links indicate which actors are involved in which use cases. Our diagram will involve the **Agent** in three use cases. But the **Passenger** will participate in only two and the **Finance Officer** in only one.

Step: Create some communication links:

1. Between **Passenger** and **Make a Reservation**
2. Between **Passenger** and **Buy a Ticket**
3. Between **Agent** and **Make a Reservation**
4. Between **Agent** and **Buy a Ticket**
5. Between **Agent** and **Check Availability**
6. Between **FinanceOfficer** and **Find Ticket Revenue**

You can use the same technique to create any linking element (communication, dependency, association, etc.). Begin creating a link by clicking the communicates button on the Diagram toolbar (↗).

A thin halo appears around potential source elements as you pass the cursor over them.	Click the source element in the Diagram pane and drag the end to the target element. Potential target elements also get the halo.
	

An actor can be a target or a source for a communication link (but not both!). You can start at the use case and stop at the actor or vice versa.

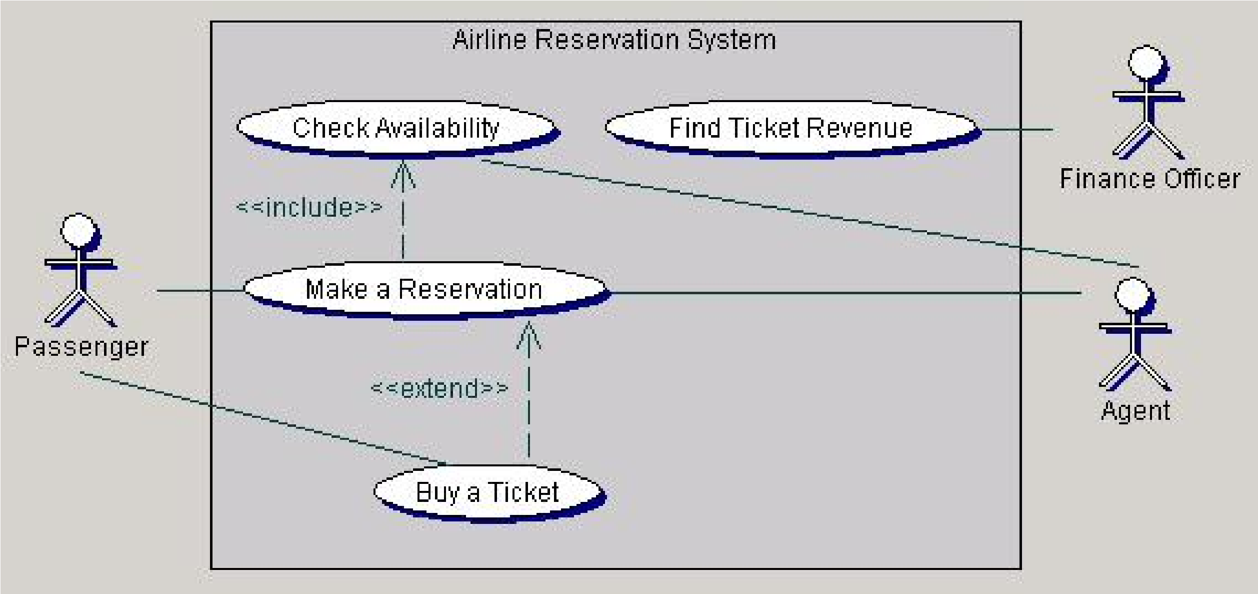
Step: Connect the use cases:

1. **Buy a Ticket** extends **Make a Reservation**.
2. **Make a Reservation** includes **Check Availability**

The Diagram toolbar has an extends button (↗) and an include button (↗). The choice of target vs. source choice is important for these links. For example, when you use extends, be sure that **Buy a Ticket** is the source and **Make a Reservation** is the target.

Step: Make sure all use cases are inside the system boundary. Drag the use cases and resize the boundary as needed.

Below is a snapshot of our diagram after we completed all the steps.



You may have to rearrange the elements to make your diagram look ours. Move them around as you see fit. You can drag them individually, or you can select several to move at the same time.

Removing diagram elements

Step: Corrupt your Use Case diagram with these steps:

1. Create a new use case (any name will do).
2. Make a communication link between the new use case and the **Agent**.
3. Create an extends from **Check Availability** to the new use case.

Then move the new use case around several times to observe how Together handles the rearrangement.

Your diagram should be a mess by now, and you need to get it back as it was before.

Together's undo button (↶) on the Main toolbar is a first line of defense in dealing with mistakes. But when the undo stack is high, the undo button may not be the best way to go.

Step: Get rid of the new (corrupt) use case.

An easy way to get rid of this use case is to delete it. Select it in the diagram and press the Delete key. (Alternatively, select Delete from the element's speedmenu.) When you do, the bad use case will disappear along with all of its links. There's no need to remove the links separately.

Tips and Tricks

- There are three easy ways to rename an actor, use case, or system boundary.

    - Double click on the diagram element to bring up the in-place editor.
    - Use the Rename option on the element's speedmenu.
    - Change the name on the Properties tab of the Properties Inspector for the element.
- To resize a use case, actor, or system boundary in a diagram, select it, then drag on one of its corner handles. To reshape drag on a side or top/bottom handle.
- If you make a mistake, use the undo button. Ctrl+Z is a keyboard shortcut for undo. Ctrl+Y is the keyboard shortcut for redo. (The **misc.config** file sets the default undo buffer size to 4096 KB. You can change that by editing the file.)

← PREVIOUS | TOP ↑ | NEXT → | START HOME

Last Revised: Fri, Mar 30, 2001

Together Tutorial
Part 3: Business Rules and Activity Diagrams

Business rules constitute some of the requirements of a problem. In our previous discussion of use cases, we limited the required features to these four:

- Make a reservation.
- Buy a ticket.
- Determine if a flight has room for more reservations.
- Find the total ticket revenue for a flight.

In this section, we will examine the details of what it means to "Make a reservation" in the context of activity diagrams.

Contents:

Starting activity diagrams based on business rules

How can you "Make a reservation?" Our airline uses this (admittedly naive) business rule:

> **You can make a flight reservation if the number of tickets sold so far for the flight does not exceed the capacity of the airplane.**

It's time to focus on exactly what happens when a reservation is requested.

Step: Create a new Activity diagram in the **ProblemDomain** package and name it **Request Reservation**.

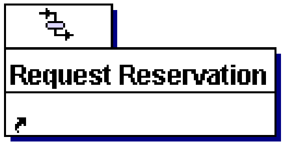
Begin this step with the New diagram icon ( ) on the main toolbar.

You'll have to enter the name in the textfield at the upper right of the dialog window. (If you didn't begin from the package diagram, select **ProblemDomain** from the Package pick list.)

Together will be able to use the description you fill in when it generates documentation.

☑ Include in current diagram

If you clicked include in current diagram, the **ProblemDomain** package diagram should show a node that is a shortcut to the activity diagram.



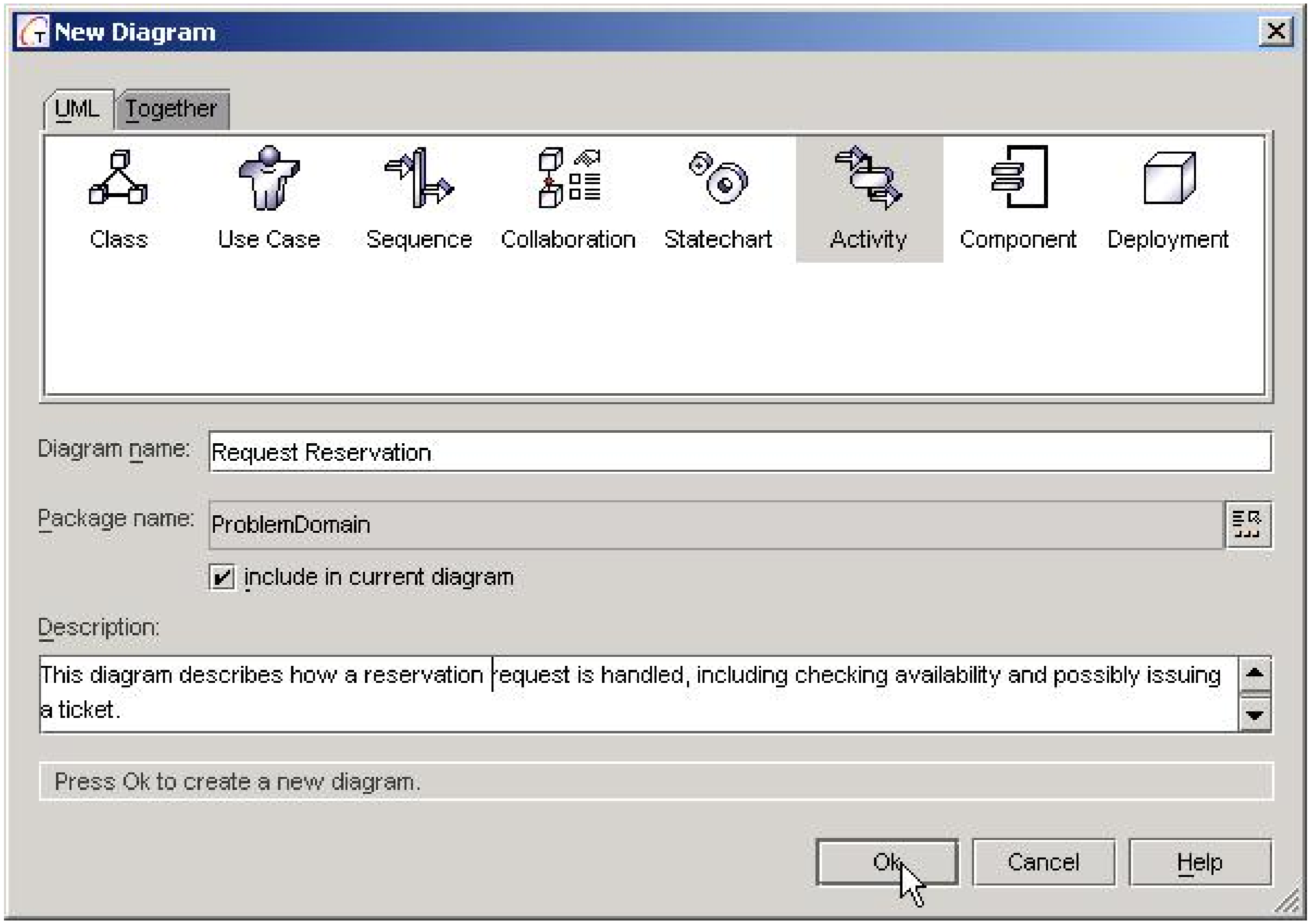


Diagram name: Request Reservation

Package name: ProblemDomain

☑ include in current diagram

Description:

This diagram describes how a reservation request is handled, including checking availability and possibly issuing a ticket.

Press Ok to create a new diagram.

You will find the following eight activity diagram toolbar buttons useful for the steps on this page.

▤	Swimlane	↗	Transition
●	Start	⇊	Horizontal fork
◉	Stop	⇉	Vertical fork
▭	Activity	◇	Decision

Organizing activities, start and stop states with swimlanes

Let's divide making a request into three pieces.

- **Airplane/Flight Description** for the capacity of the airplane
- **Flight Reservations** for the list of reservations on this flight
- **Reservation/Ticket Services** for creating reservations and issues tickets

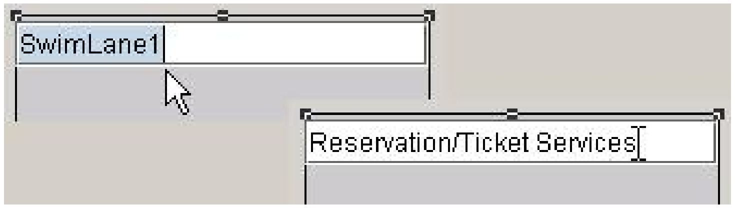
Keep in mind that the swimlanes are often not associated with classes or objects -- especially since business modeling frequently precedes class diagram design!

Step: Make three swimlanes in the diagram.

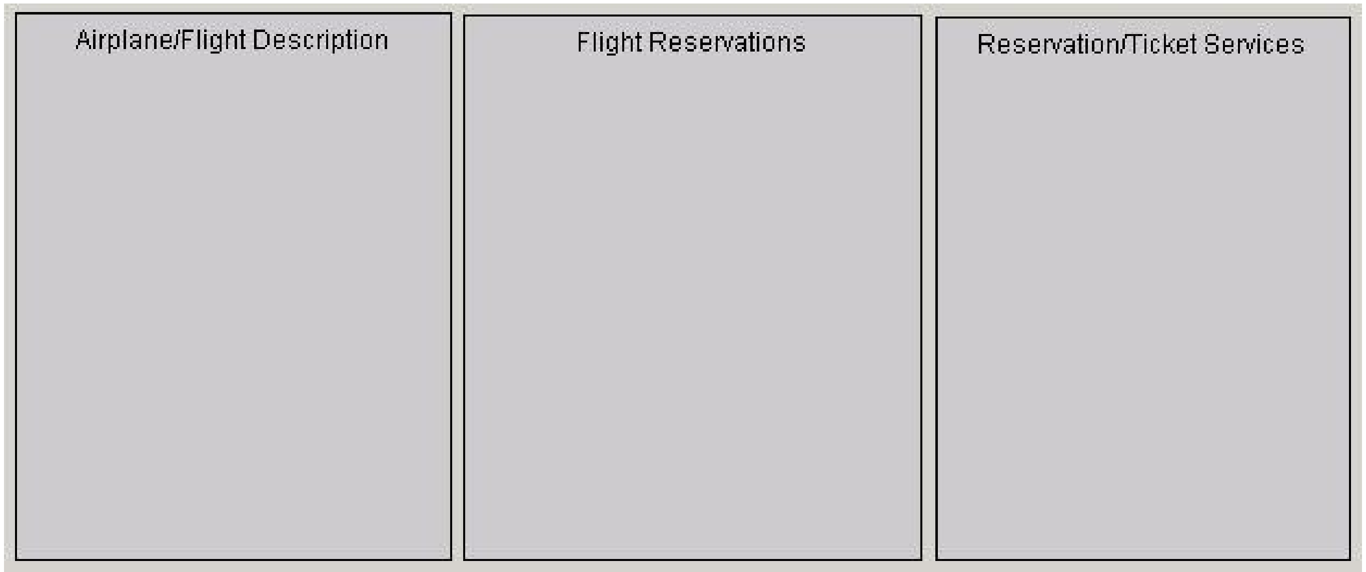
1. **Airplane/Flight Description** on the left
2. **Flight Reservations** in the middle
3. **Reservation and Ticket Services** on the right.

To create a swimlane, click the swimlane button (▤) on the Diagram toolbar, then click the diagram.

You can change a swimlane name by clicking on the name to bring up the in-place editor.



The resulting diagram is pretty simple at this point.



Step: Put a start state at the top of the activity diagram (above the swimlanes) and put a stop state below the swimlanes.

This step is easy: click start-button then click diagram; click stop-button then click diagram.

Creating activities and transitions

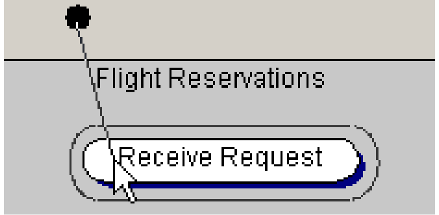
The initial activity for the activity diagram will be receiving a reservation request.

Step: Create an activity named **Receive request** and put it inside the **Flight Reservations** swimlane. Link the start to the activity with a transition.

Creating activities on activity diagrams is analogous to creating use cases on use case diagrams. Start with the activity button on the toolbar (▭). You can move activities around. You can edit them with the in-place editor.

Activity diagram transitions are analogous to use case diagram communications. When you click the toolbar transition button (↗), Together puts halos around potential sources and targets for the transition as you pass the cursor them.

At the right is a snapshot of making the transition from the start point to the activity.



Step: Create a five more activities:

1. **Get capacity (cap) -- Airplane/Flight Description** swimlane
2. **Get #tickets -- Flight Reservations** swimlane
3. **Create reservation-- Flight Reservations** swimlane
4. **Refuse request -- Flight Reservations** swimlane
5. **Issue ticket -- Reservation/Ticket Services** swimlane

The Ctrl+click technique works for creating multiple activities the same as for creating multiple use cases. When you Ctrl+click the button to place a new activity on the diagram, you can edit the activity in place and then click again to create another new activity.

Once you create an activity, you can drag it to any swimlane (or even outside swimlanes entirely).

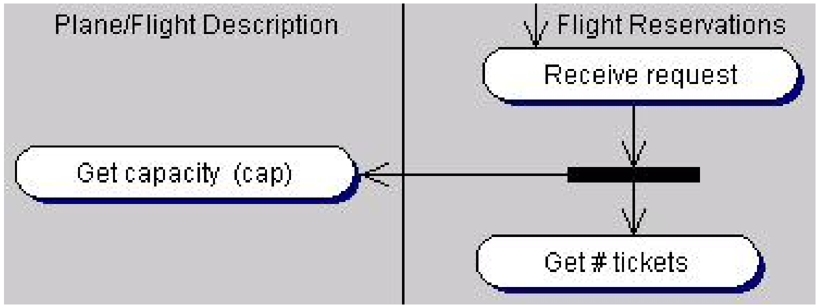
Changing flow of control with forks, joins, and decisions

Before our airline can make a reservation, it checks to see if the flight has room. That's where the business rule comes in. **Get capacity** and **Get #tickets** can be performed in either order. But they both have to be completed before the remaining activities can begin.

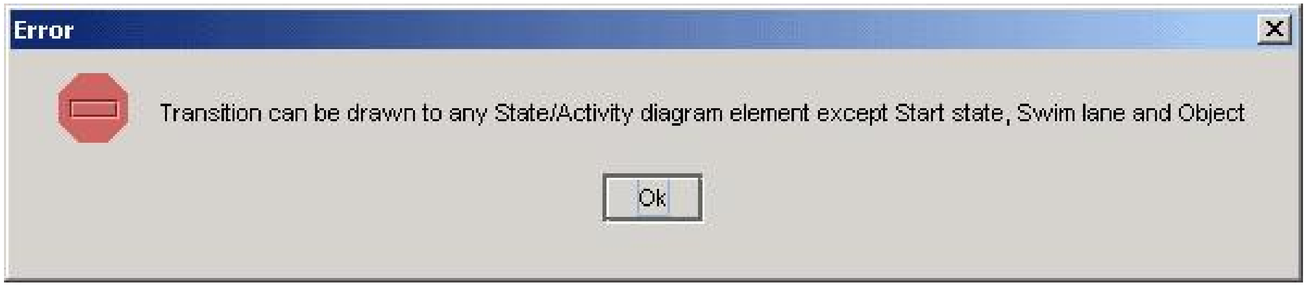
Step: Create a fork. Make a transition from **Receive request** to the fork. Then make transitions from the fork to **Get capacity** and to **Get #tickets**.

The fork buttons on the diagram toolbar give a choice of horizontal forks ( ) or vertical forks ( ). Which you choose depends only on how you want the diagram to look.

A fork can be a source or a target of a transition ( ).



Be sure to look for the halo when you draw a transition to a fork. Forks are so slim that it is easy to miss a target fork and land on a swimlane instead. If you try to end a transition on a diagram entity that is not a valid target, Together writes a red error message in the Message pane, and it displays an error box like the one below.



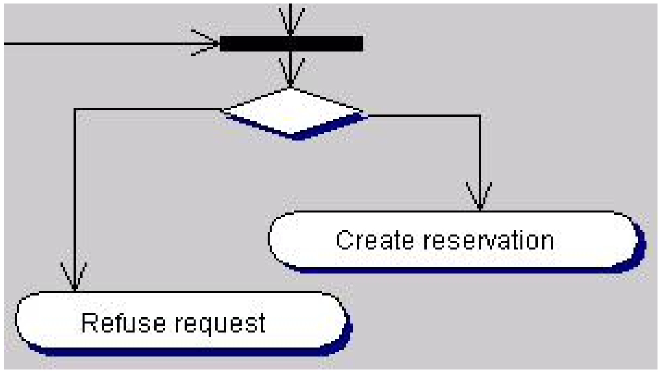
Transition can be drawn to any State/Activity diagram element except Start state, Swim lane and Object

Step: Create a join. Then make transitions from the **Get capacity** and to **Get #tickets** to the join.

The join button is the same as the fork button. You can choose either the horizontal or vertical version. Again, which you choose depends only on how you want the diagram to look.

Step: Make a decision node to compare the number of tickets to the capacity of the airplane. Make a transition from the join to the decision. Then make a transition from the decision to **Create reservation** and another transition to **Refuse request**.

The decision button is the diamond (◇) on the diagram toolbar.



To get the snapshot on the left, we set our Diagram Options to show rectilinear links. The options can be set from the Main menu under **Options|Diagram**.

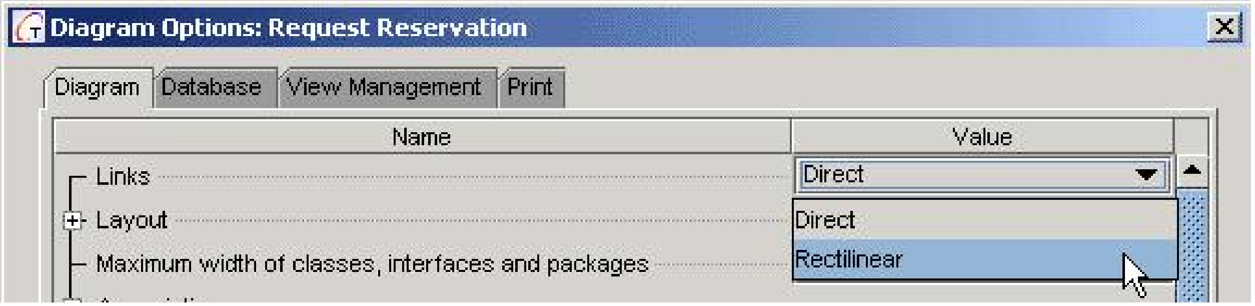


Diagram Options: Request Reservation

Name	Value
Links	Direct
Layout	Direct
Maximum width of classes, interfaces and packages	Rectilinear

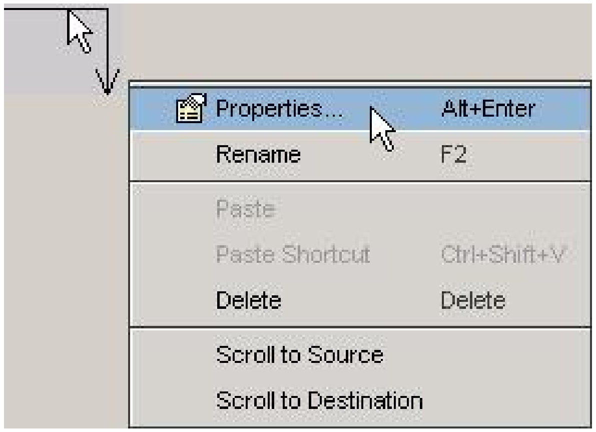
The transitions out of the decision will be complete when they're labeled with guard conditions to indicate which transition applies.

Step: Put guard conditions on the transitions out of the decision as follows:

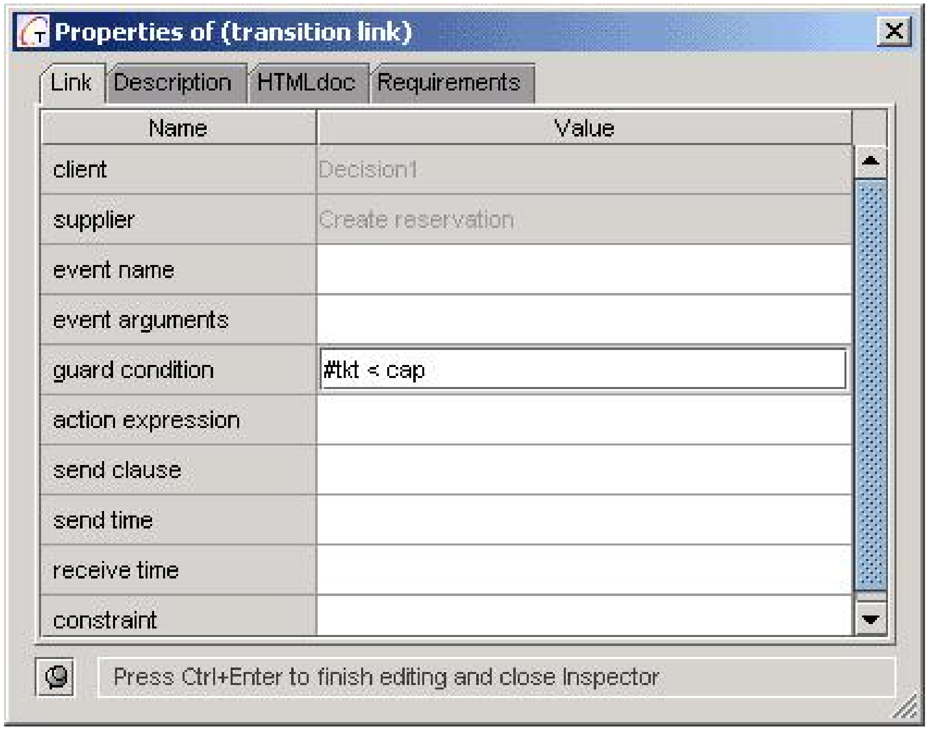
- #tkt < c on the transition to **Create Reservation**
- #tkt >= c on the transition to **Refuse request**

Set the requirement type to business rule. And fill in a descriptive comment to be used later for project documentation.

Transitions have inspectors that you can access from their speedmenus. Right click on the transition line and you'll get the properties inspector at the right.



The link properties inspector has a Link tab with a textfield for the guard condition. The Req tab has a textfield for business rules. The Description tab has a textbox for descriptions.


Properties of (transition link)

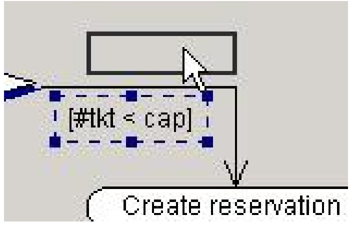
Name	Value
client	Decision1
supplier	Create reservation
event name	
event arguments	
guard condition	#tkt < cap
action expression	
send clause	
send time	
receive time	
constraint	

Press Ctrl+Enter to finish editing and close Inspector

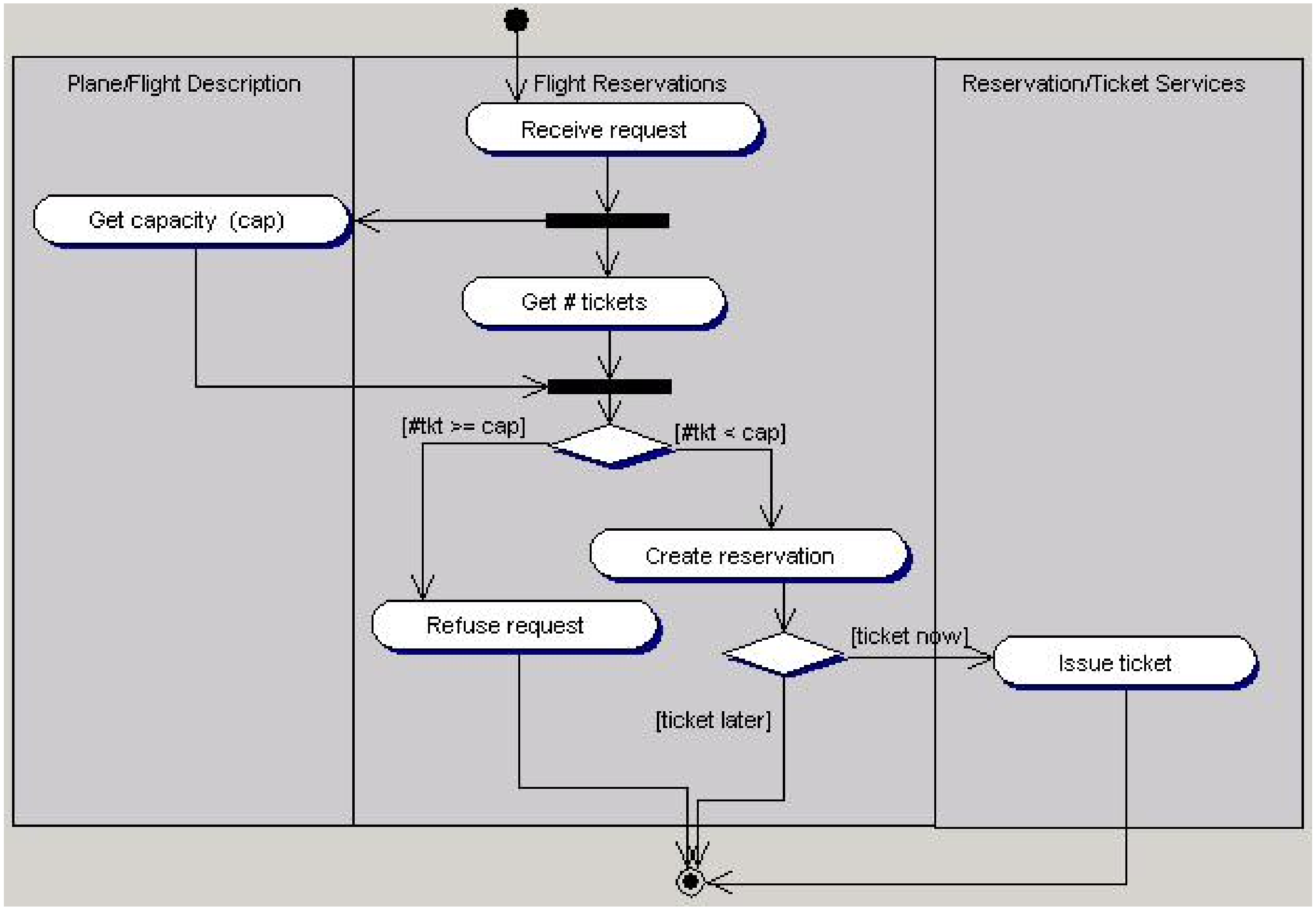
Step: Put in a second decision. Make four transitions:

1. from **Create reservation** to the new decision.
2. from the new decision to **Issue ticket**. Put a guard condition on the transition: **ticket now**
3. from the new decision to **stop**. Put a guard condition on the transition: **ticket later**
4. from **Refuse request** to **stop**.
5. from **Issue ticket** to **stop**.

Guard conditions appear on the diagram in square brackets. You can change a diagram by dragging guard conditions or activities.



Below is a picture of our completed activity diagram.



Tips and Tricks

- Activity diagrams are fancy flow charts. Use them to spell out the details of potentially complicated or arcane business rules.
- Together makes no direct connection between code and activity diagrams. Activity diagrams are useful for sketching out the flow of activities. They need not spell out exact messages, message sequencing, or control structures.
- When Together cannot determine where you want a transition to end, it will put up a "Choose Destination" dialog box giving you a choice of possible endpoints.
- The **Options|Diagram** command on the Main menu lets you change the links from rectilinear to direct.
- Almost all diagram elements have speedmenus. Access to a speedmenu is always the same -- right click on the element.
- The speedmenu for a transition can be accessed through the transition or any diagram annotation on the transition (such as a guard condition).

← PREVIOUS | TOP ↑ | NEXT → | START HOME

Last revised: Thu, Apr 12, 2001

Together Tutorial
Part 4: Diagrams and Classes

We have already created a new project and organized our work with packages. And we've sketched out some requirements with use cases. In this section, we'll put some flesh on the problem domain package by creating classes.

Some of the tasks for this section are designed to illustrate Together's LiveSource$^{\text{TM}}$ always-in-sync technology. It's one of the hallmarks of Together that never fails to impress. Have fun!

Contents:

Creating new classes

Here are the initial requirements of the **airline** project.

- Make a reservation.
- Buy a ticket.
- Determine if a flight has room for more reservations.
- Find the total ticket revenue for a flight,

From those requirements, we came up with five problem domain classes. We've listed them here along with some of possible attributes and operations. This is just a start. As we go along, we may find the need for additional classes and members.

Class	**FlightDescription**	**ScheduledFlight**	**Reservation**	**Ticket**	**Agent**
Attributes	departureTime arrivalTime origin destination capacity	date	name	basePrice	name

Methods	getCapacity()
	setCapacity()
		makeReservation()
		numberOfTickets()
			ticketPurchased()
			calcPrice()
				calcPrice()	makeReservation()

You'll want to begin your work by bringing the **ProblemDomain** package into focus in the Diagram pane.

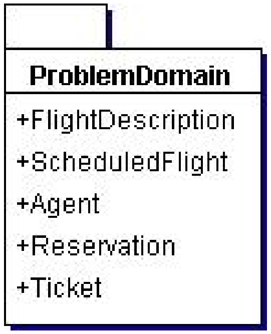
Step: Create five classes in the **ProblemDomain** package: **FlightDescription**, **ScheduledFlight**, **Reservation**, **Ticket**, and **Agent**.

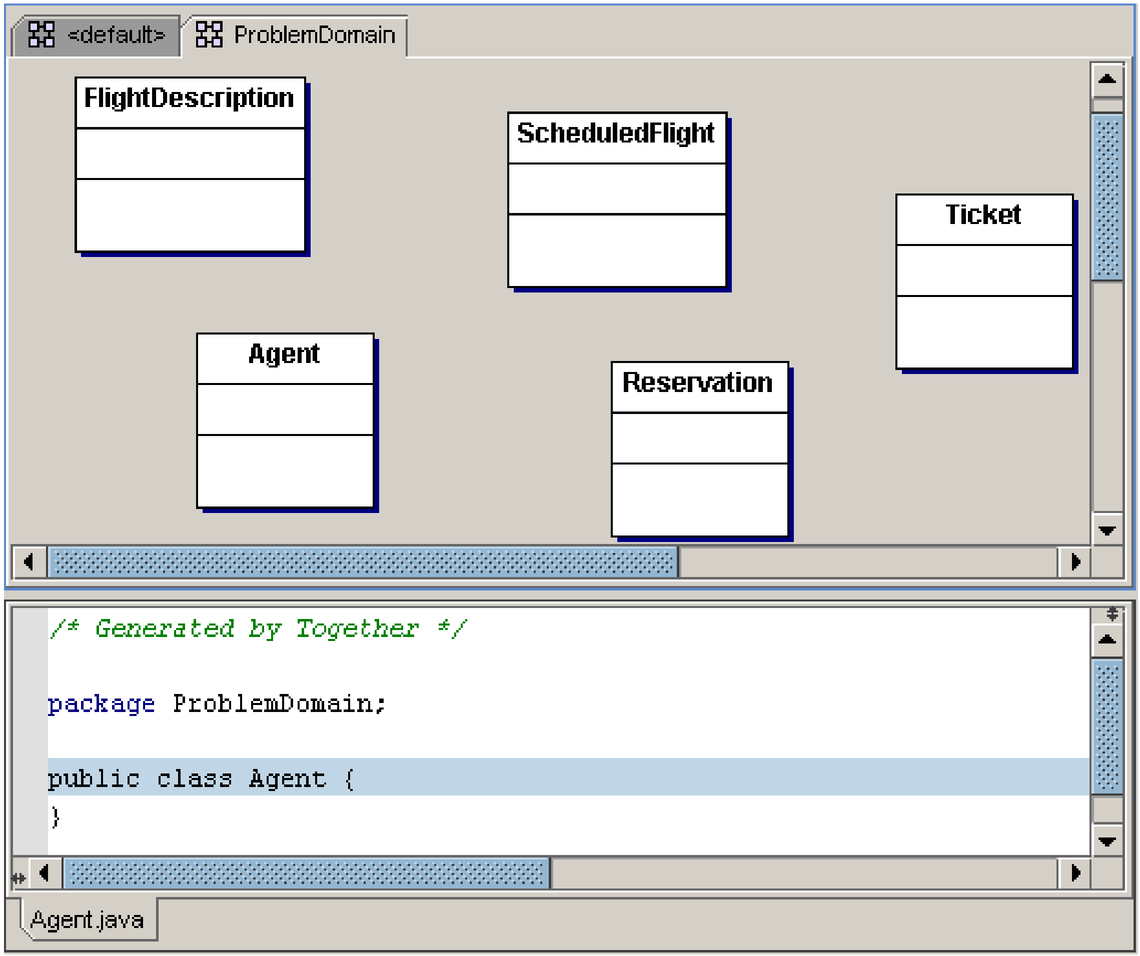
Ctrl+click the class button on the Diagram toolbar (▤) to create multiple classes. Name the classes in the diagram as you go along. There's no need to write any class declarations -- Together generates them automatically.

As you create the classes in the Diagram pane, the Editor pane displays the new code.

If you go to the <default> diagram, you'll see that the **ProblemDomain** package now shows the new classes. Changes in one diagram often create changes in related diagrams.

Below is a picture of the **ProblemDomain** package node. The + beside each name indicates that the class is public.





```
/* Generated by Together */

package ProblemDomain;

public class Agent {
}
```

Together has now generated source files for the classes in the physical directory of the package. The Explorer Model tab tracks the new classes as part of the current project.

Note: While this project won't have any inner classes, they're easy to create. Simply drag one class inside another. Or you can click the class button from the toolbar and then click inside the outer class on the Diagram pane.

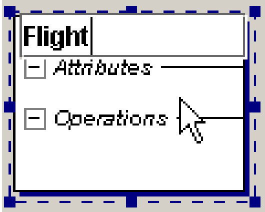
LiveSource^{TM} always-in-sync technology

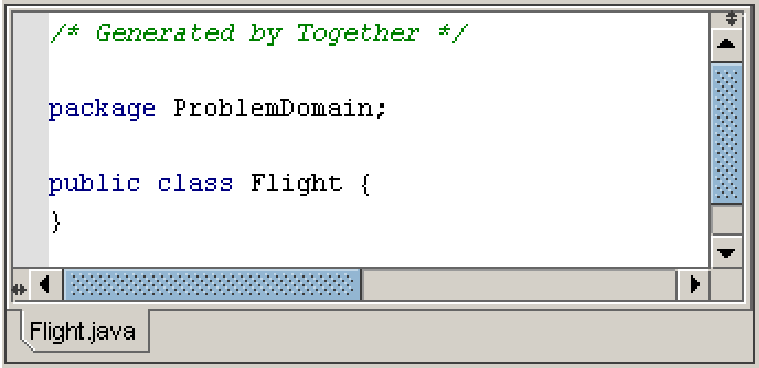
Together uses your source code to construct its diagrams. It does not keep a repository. When you make legitimate code changes, the diagram shows those changes... and vice versa. In this section and the next, we'll show you how to change some elements of your project just to see Together's LiveSource^{TM} technology at work.

Step: Use the Diagram pane to rename the **ScheduledFlight** class to **Flight**.

Use the class speedmenu to change a class name. Or simply double-click on the name in the Diagram pane.

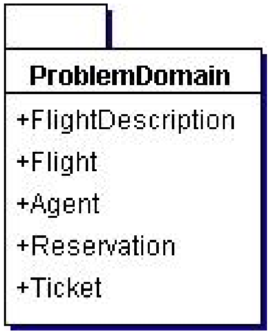
Together updates code, filename, and diagram.





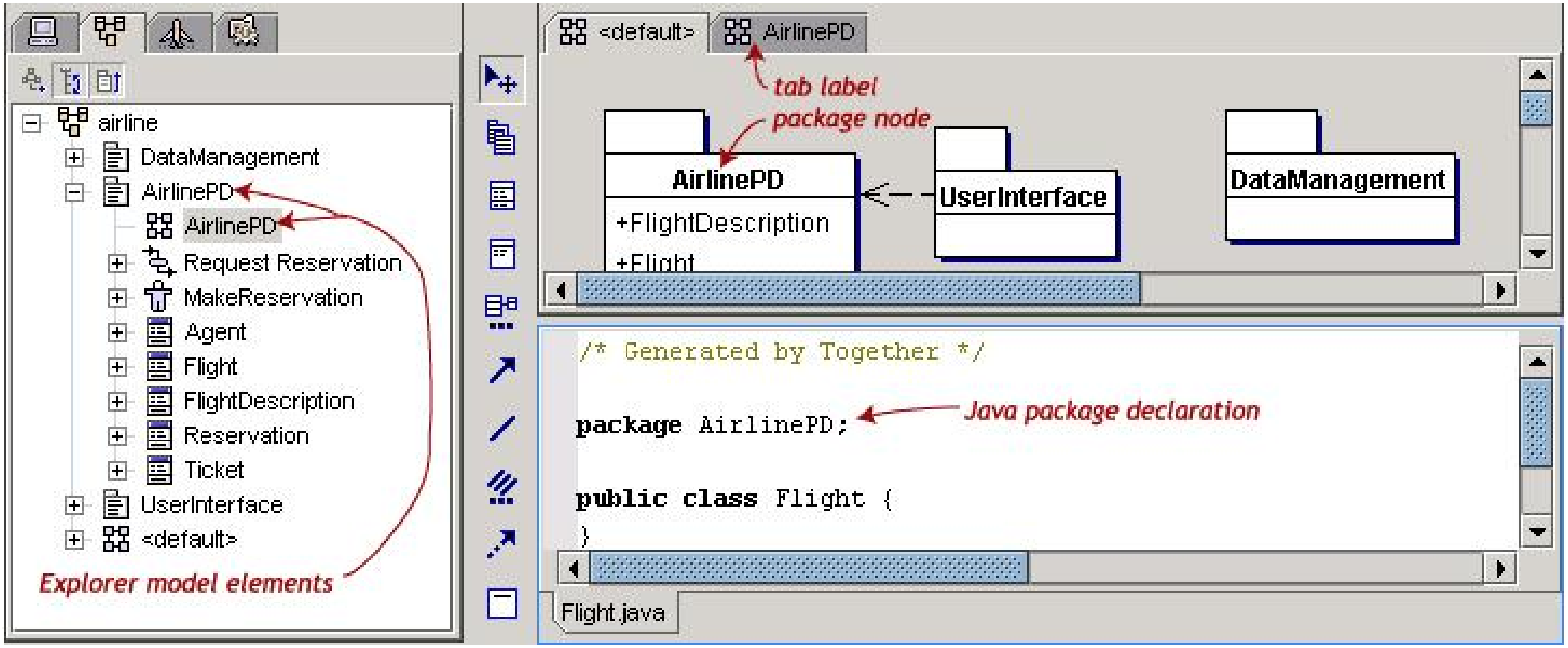
```
/* Generated by Together */

package ProblemDomain;

public class Flight {
}
```

If you open the <default> diagram, you'll see that Together has updated the class name in the **ProblemDomain** package node.



Step: Use the **ProblemDomain** speedmenu in the <default> diagram to rename **ProblemDomain** to **AirlinePD.**

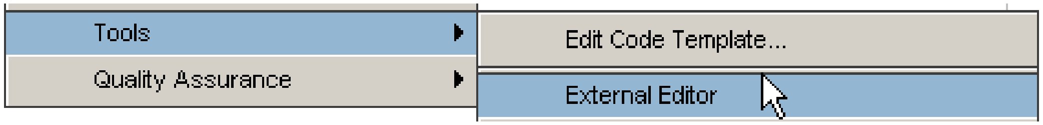
Use the **ProblemDomain** speedmenu in the <default> diagram to rename the package. The snapshot below points out four changes that Together makes as a result. The Explorer Directory tab shows that the physical directory is renamed as well.



```
/* Generated by Together */

package AirlinePD;

public class Flight {
}
```

Editing source code outside of Together

Together does not force working in a particular fashion. You can edit all of your source code in your favorite editor at the same time you manage your project within Together.



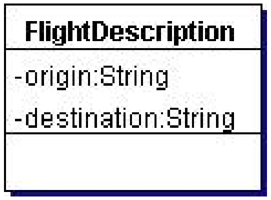
The **Tools|External Editor** command on the class speedmenu gives a quick way to open the source file in an external editor.

You can get to the class speedmenu through the Diagram pane or Explorer pane.

Step: Open **FlightDescription.java** in an external editor. Add two **String** fields, **origin** and **destination**, and then save the file in the editor.

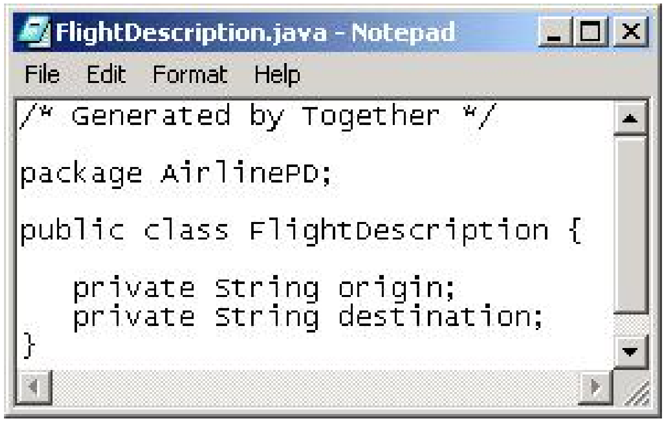
Notepad is the default external editor for Windows. (That's easy to change in the Project options. Look at the Tools menu.)

We opened **FlightDescription.java** in Notepad and entered the **origin** and **destination** declarations.



When we saved our source file in Notebook, Together updated the diagram. Together also updated the source file in its Editor pane view.

Together updates its Editor and Diagram views when the time stamp on a source file changes.



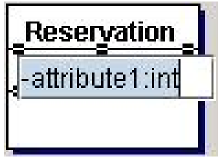
```
/* Generated by Together */

package AirlinePD;

public class FlightDescription {

   private String origin;
   private String destination;
}
```

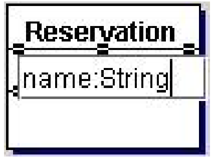
Adding attributes

It is often quickest to add members to a class through its Diagram pane node.

Step: Go to the Diagram pane to add a **name** (type **String**) to **Reservation**.

To add a new attribute to a class, select the class in the Diagram pane and choose **New|Attribute** from the class speedmenu. (Or simply use the keyboard shortcut, Ctrl+A.)

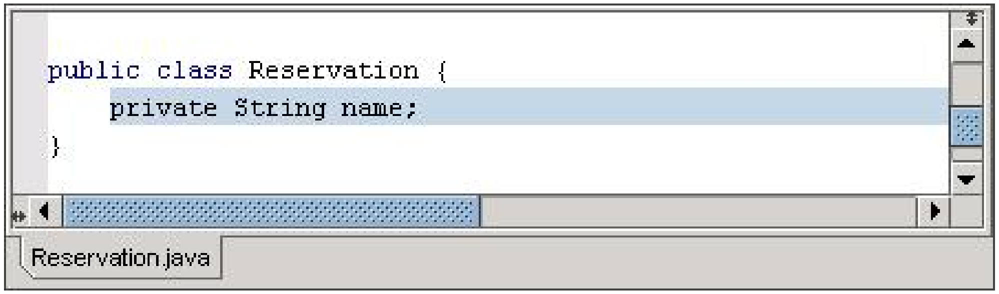




The new member gets a default name (attribute), type (int), and visibility (private). You can immediately change all those by in-place editing on the diagram node. (Selecting the member on the diagram then clicking on it activates the in-place editor.)

If you fill in just a new name and press Enter, then the name changes but the default attribute type (or operation return type) and visibility level remain.

Of course, Together keeps diagram and code in sync, automatically adding the declaration to the source code. The snapshot below shows the results in the Editor pane.



```
public class Reservation {
    private String name;
}
```

There are several ways to add attributes to a class through the Diagram pane.

- If a class already has an attribute and you want to add more, select the attribute and then press the Insert key.
- If a class has an attribute that you want to duplicate within class, select clone from the attribute speedmenu and then edit the result.

- Select an attribute and copy it via its speedmenu. Then paste it into another class using the speedmenu for that class.
- Move an attribute from one class to another using drag-and-drop. When you drag the attribute to a valid destination class, you'll see a blue halo around the class.
- Copy an attribute from one class to another through Ctrl+drag-and-drop.

You can drag-and-drop attributes within a class to reorder them.

Step: Edit three properties of the **name** attribute of the **Reservation** class.

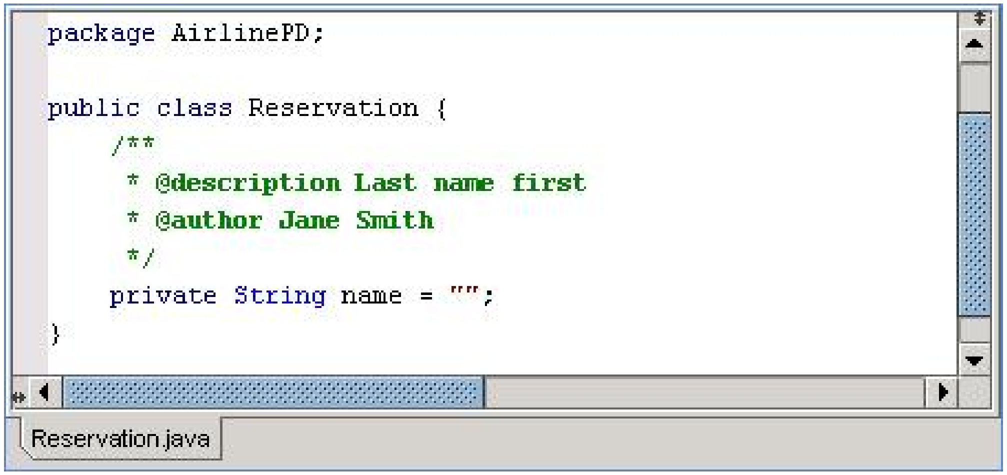
1. Initialize the value to the null string.
2. Set the "requirement description" to Last name first.
3. Set the author to your name.

The snapshot below shows that editing attributes can generate Javadoc comments as well as code.

You can edit class member properties through their inspectors. (You need to select the class member and not the entire class in this case.)

- Set the initial value of an attribute on the Properties tab of the inspector.
- Set requirement types and descriptions on the Req tab.

Once you've completed the edits, press Ctrl+Enter. This saves the changes and closes the inspector.



```
package AirlinePD;

public class Reservation {
    /**
     * @description Last name first
     * @author Jane Smith
     */
    private String name = "";
}
```

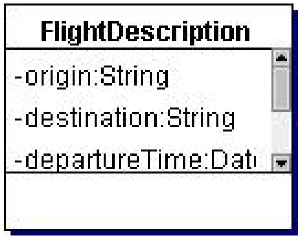
Step: Put additional attributes in the five **AirlinePD** classes. Use the Diagram pane rather than the editor to enter the new attributes.

1. **Agent**: **name** (copy it from the **Reservation** class)
2. **FlightDescription**: **departureTime**, **arrivalTime** (Wait on **capacity** for the next section.)
3. **Flight**: **date**
4. **Ticket**: **basePrice**

Together puts vertical scrollbars on class nodes in the Diagram pane that are too small to show all of their members.

You can resize a class node by grabbing the handles around its border and dragging. Invoking "Actual Size" from its speedmenu auto-resizes the class node and displays all members.

Want to change the ordering of class members? Use drag-and-drop to reposition them.



Adding operations

The same techniques for adding new attributes apply to adding new operations. The keyboard shortcut is Ctrl+O.

Step: Add a new void operation named **makeReservation** to the **Flight** class. Give the operation two parameters: a **String** for the name of the passenger and an **int** parameter for the kind of ticket**.**

The in-place editor for an operation on a diagram node takes input in UML style (type follows name) or Java/C++ style (name follows type). The table below illustrates the difference.

Style	Format	Example
UML	*name(parameters):type*	myMethod(myParameter:int):double
Java/C++	*type name(parameters)*	double myMethod(int myParameter)

The default visibility for attributes is private. The default visibility for operations is public. You can change the visibility of a member with the in-place diagram node editor, in the inspector, or in the source code.

Step: Use the Diagram pane to copy **makeReservation()** from **Flight** to **Agent**.

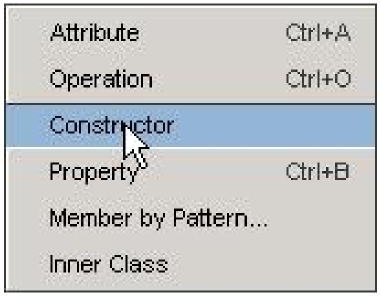
To copy an operation from one class to another, use Ctrl+drag-and-drop or "Copy" from the operation speedmenu and "Paste" from the speedmenu of the target class. The Diagram pane copy duplicates the entire operation from one class to another, including its body.

Step: Add these operations to the **Reservation** class.

- constructor with a **String** parameter and an **int** parameter
- **ticketPurchased()** returns a **boolean**
- **calcPrice()** returns a **double**

The class speedmenu has a **New|Constructor** command that you can use to create the **Reservation** constructor.

You will eventually use two other items on this speedmenu: **New|Property** and **New|Member by Pattern**. (The snapshot shows that it is possible to create inner classes via the speedmenu. You can do the same thing by dragging a class inside the intended outer class.)



Step: Add these operations to the **Flight**, **Ticket**, and **FlightDescription** classes.

- **Flight - numberOfTickets()** returns an **int**
- **Ticket - calcPrice()**. Copy it from the **Reservation** class.

Together treats attributes with corresponding getter and setter operations as properties.

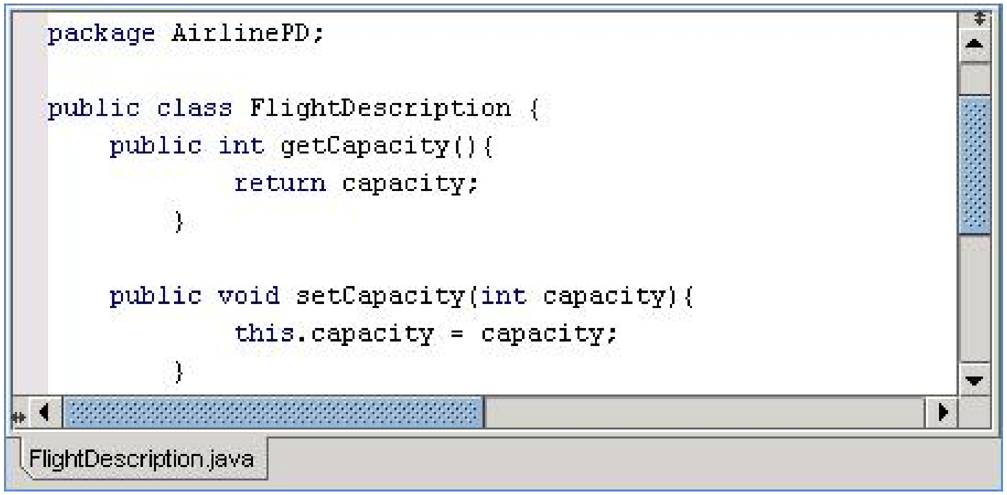
Step: Use the **FlightDescription** speedmenu to add **capacity** as a property.

Begin this step by selecting **Property** from the class speedmenu.

When you create a property, Together automatically creates getter and setter methods along with the code for their bodies.

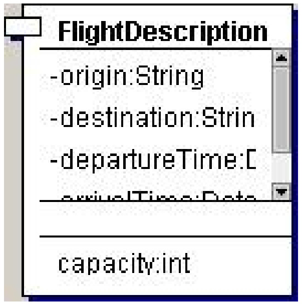
The snapshot here shows the result of changing the default name from **property** to **capacity**. Together makes several code changes:

- The name of the getter changes to **getCapacity**.
- The return statement returns **capacity** (rather than **property**).
- The name of the setter changes to **setCapacity**.
- The body of the setter assigns to **this.capacity**.



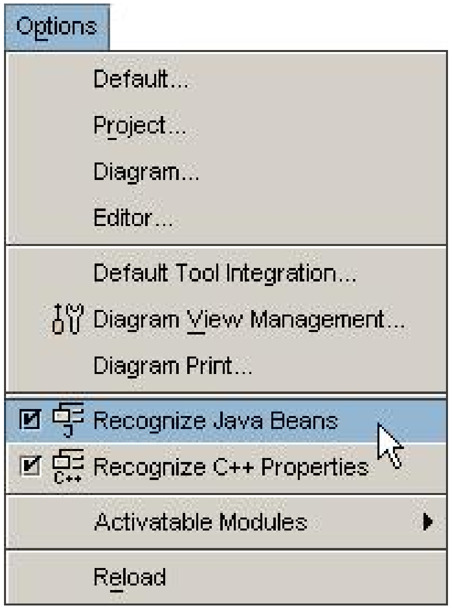
```
package AirlinePD;

public class FlightDescription {
    public int getCapacity(){
            return capacity;
        }

    public void setCapacity(int capacity){
            this.capacity = capacity;
        }
```

Together recognizes classes with getters or setters as JavaBeans.

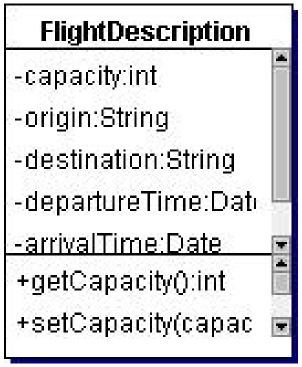


A JavaBean diagram node has a characteristic tab on the upper left. Getters and setters do not appear among the operations. And the property attributes move from the attribute compartment to a properties compartment at the bottom of the class diagram.

The Options menu has a check box to turn on/off JavaBean recognition.



When you check **Recognize JavaBeans** off, the properties appear as attributes and the getters and setters appear as operations.



Properties and attributes have different speedmenus. Before trying the next step, make sure that **Recognize JavaBeans** is turned off.

Step: Use the **capacity** speedmenu to initialize it to 50.

Right click on a class member to get the speedmenu for the member (not the speedmenu for the class). The Properties tab of an attribute speedmenu has an initial value field.

Changing class properties (aka color me pink -- or blue or green or yellow)

Step: Change the **Ticket** class:

1. Make **Ticket** an abstract class.
2. Make **calcPrice()** an abstract method.

You can make a class abstract by using its speedmenu or its property inspector. Check the Abstract box on. Use the same procedure to make a method abstract.

Your **Ticket** node should now display its name and method in italics (as specified by the UML).

The class inspector can add a huge variety of model richness. The inspector is organized by tabs. Together translates some of the tab items into source code (such as determining class visibility or setting the class node to be an interface). It translates other tab items into Javadoc comments within the source code (such as those on the Javadoc and Description tabs).

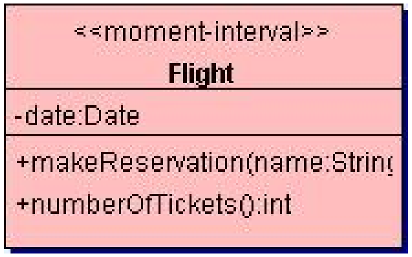
Step: Give **Flight** the "moment-interval" stereotype.

The Properties tab of the class inspector has a stereotype field with a pull-down list. Moment-interval is one of the color stereotypes featured in *Java Modeling in Color with UML* by Peter Coad et al.

Together colors a node with a "moment-interval" stereotype pink. Your **Flight** node should be pink.

The pink color is a characteristic of both "moment-interval" and "moment-interval-detail" stereotypes. The "role" stereotype is yellow; "party", "place", and "thing" stereotypes are green; and the "description" stereotype is blue.

Together does not limit stereotypes to the choices in the pull-down list ... you can type in anything you want. And you can customize your stereotype to your own color via the class inspector View tab.


<<moment-interval>>

Flight

-date:Date

+makeReservation(name:Strin(

+numberOfTickets():int

Step: Add color stereotypes to the rest of the classes:

- **FlightDescription**: description
- **Reservation**: mi-detail (moment-interval detail)
- **Ticket**: thing
- **Agent**: role

When you finish with that last step, your diagram ought to come alive in four colors!

Tips and Tricks

- Keyboard shortcuts make adding new class members speedy -- Ctrl+A for attributes and Ctrl+O for operations.
- You can move a member from one class to another merely by dragging it from one class node to the other.
- You can copy a member from one class to another by Ctrl+dragging it.
- Attributes and operations appear in a diagram node in the same order that they appear relative to each other in the code. You can change the position of a member by dragging it within the node.
- If you change a class or package name in a diagram, Together will change the file names and package statements accordingly. If you change a class name

in the editor instead, Together assumes you are the expert responsible for changing the corresponding file names and package statements.

- You can initialize an attribute by editing the source code or by specifying an initial value in the attribute's properties inspector.
- Most activities in Together can be done in different ways. For example, you can enter a property through the class speedmenu, by adding the attribute and then the getter and setter operations, or by editing the source code directly.

← PREVIOUS | TOP ↑ | NEXT → | START HOME

Last revised: Thu, Apr 12, 2001

Together Tutorial
Part 5: Classes and Associations

In the previous part of the Together Tutorial, you created some classes. But it's a pretty weak class diagram that has no associations. In this part, you will remedy that. You will create associations linking the five classes that you've created so far. You will label some of the associations and put on multiplicities.

At the end of this part of the Tutorial, you will create an interface and two subclasses. And you'll connect them to existing classes via generalization and implementation links.

Contents:

Creating associations

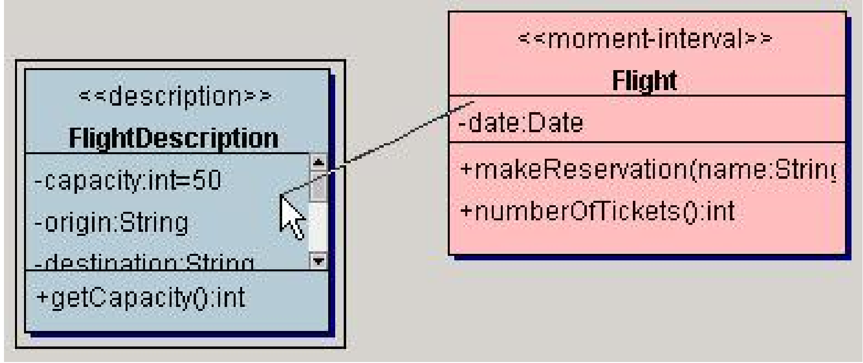
You'll be working in the **AirlinePD** package. Open its diagram in the Diagram pane.

Step: Create an association from **Flight** to **FlightDescription**.

Begin this step by clicking the association button (/) on the Diagram toolbar. Associations are handled the same way as linking elements in other diagrams. As you pass the cursor over valid client (start) and supplier (end) nodes, Together puts a halo around them.

Drag the cursor from **Flight** to **FlightDescription** and release. The Diagram pane will show the new association as a blue line.

If you look in the Editor after creating the association, you'll see that **Flight** has member named **lnkFlightDescription**.



Note that **lnkFlightDescription** does not appear in the **Flight** node of the Diagram pane. Together considers any attribute whose name begins with **lnk** to be an "Automatic" link, and it will not show it in a class node of the Diagram pane unless you reset the diagram options. (If you want the new link member to show as an attribute, you can change its name to begin with something besides **lnk**.)

Step: Create two more associations:

1. From **Reservation** to **Ticket**
2. From **Agent** to **Flight**

Together gives a choice of link display:

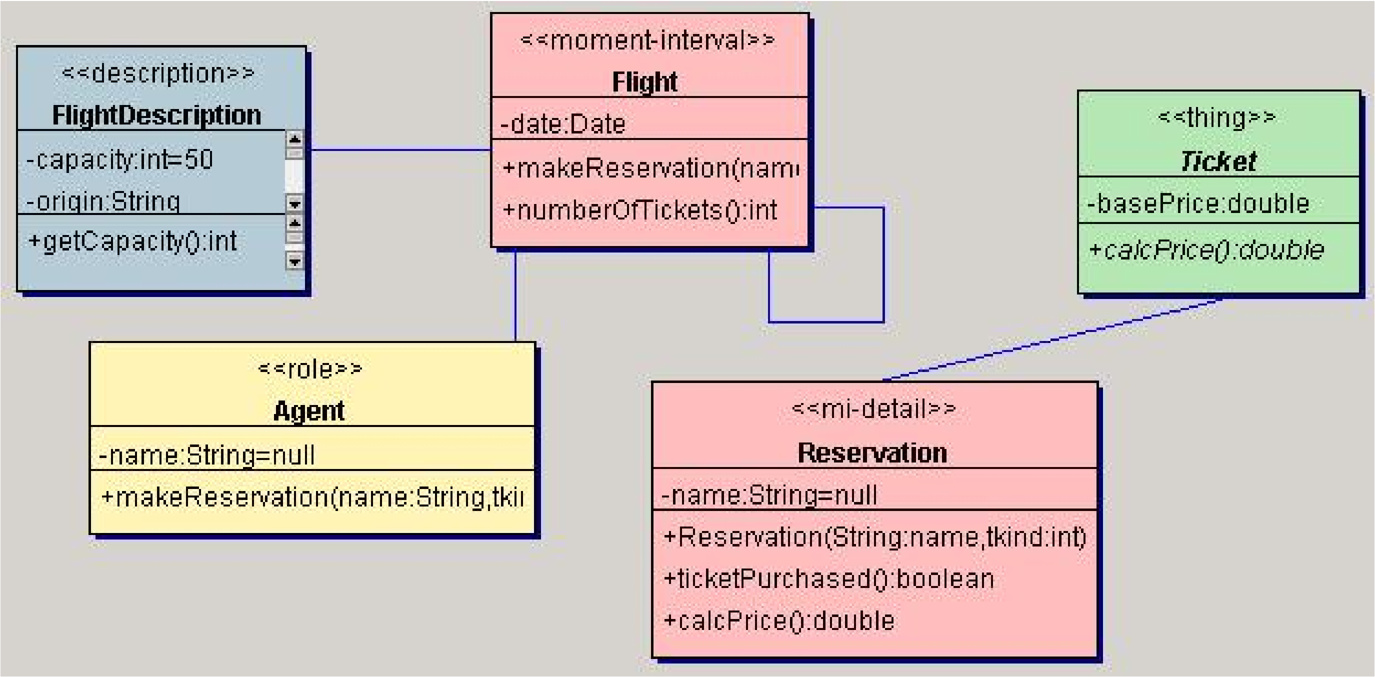
- rectilinear: the link is a sequence of horizontal and vertical line segments.
- direct: the link is a line segment, but it may be slanted. (The link may also be a sequence of line segments with any slope).

Link displays are set on the diagram options, accessible via **Options|Diagram** on the Main menu.

Most links go from one class to another. But some can be self-links, starting and ending at the same class. You can put a self-link in our model to distinguish between a flight plan and an actual flight.

Step: Make the diagram link display to be Direct. Then draw a self-link on **Flight**.

Below is a snapshot of the diagram so far.



Modifying associations

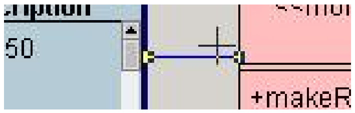
Association links are Together objects with their own speedmenus and inspectors. For many modifications, you can use either speedmenu or inspector.

Step: Put cardinalities on the link from **Flight** to **FlightDescription**.

- **0..*** at the client end (**Flight**)
- **1** at the supplier end (**FlightDescription**)

The link speedmenu varies according to which end is closer to the cursor. To target the end of a link in order to assign it a cardinality or a role, right-click on the link near that end.

The speedmenu lists the most commonly seen cardinalities. Frequently you can select the appropriate cardinality without going to the link inspector.



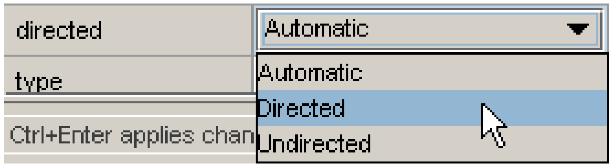
Step: Assign appropriate cardinalities to the link connecting **Reservation** and **Ticket** using this rule:

> A ticket must be associated with exactly one reservation, but some reservations may not have any tickets.

Assign cardinalities on the link from **Agent** to **Flight** using this rule:

> An agent may know about many flights; a flight may be known by many agents.

The link inspector has a rich menu for modifying links. Use it to assign roles to the link ends and to make the link directed.



directed Automatic

type

Ctrl+Enter applies chan

Step: Assign a role and **0..1** cardinalities to each end of the self link on **Flight** and make it directed. Then name the ends:

- Name the supplier role "actual."
- Name the client role "plan."

For that last step, use the inspector for the **Flight** self link.

Changing a link to an aggregation

Every **Flight** has a collection of **Reservations** rather than just a single **Reservation**.

Step: Make an aggregation from **Flight** to **Reservation**.

This is an easy step. Create the association, starting at **Flight** and ending at **Reservation**. Then bring up the speedmenu of the new association and check the Aggregation box on.

The link should now show with a diamond at the **Flight** end.

Making a new association has the potential of cluttering the diagram. You can get Together to rearrange the diagram with the diagram speedmenu (**Layout|All**). Or you can simply tweak the link directly.

When you select a link, the cursor changes to a cross (┼). Moving the cross reshapes the link. If you move the cursor to an end of the link, the cursor changes shape to a 4-ended arrow(✥). At this point, you can move the endpoint of the link to a different class.

Showing inheritance relationships

You will now extend two concrete classes from **Ticket**, which is an abstract class. You'll also make an interface for the **Agent** and **Reservation** classes.

Step: Create a class named **Coach**. Then complete these steps.

1. Link **Coach** to **Ticket** as a generalization.
2. Copy **calcPrice()** from **Ticket** to **Coach**.
3. Copy **calcPrice()** from **Coach** to **Reservation**.
4. Change the **Coach** stereotype to "thing."

Use the Generalization link button (↗) to make **Coach** extend **Ticket**. When you copy the abstract operation to a concrete class, Together will make the copy concrete in both the diagram and the source code.

Step: Make a copy of the **Coach** class and rename the copy **FirstClass**.

You can make a copy of a class using the Copy command on the class speedmenu followed by the Paste command on the diagram speedmenu. The new class is identical to the old one except for its name. All constructors are copied with the new name. All of the links that start at the original class are copied. And all Javadoc comments in the code are copied.

Step: Clean up the diagram:

- Change the links to Rectilinear.
- Make the layout inheritance horizontal.

For diagram-wide changes such as in the last step, you'll need to pull up the diagram options window. Go to **Options|Diagram** on the Main menu.

Creating and implementing interfaces

For the final task, assume that both **Agents** and **Reservations** need to be able to tell their names.

Step: Create an interface named **INamed**. Then complete these steps.

1. Give **INamed** a single operation, **getName()**, that returns a **String**.
2. Link **Agent** and **Reservation** to **INamed**.

The process for creating interfaces is identical to creating classes and packages. Just use the interface icon (▤) on the Diagram toolbar instead of the class or package icon. You can edit the name in place. Notice that Together shows names and operations of interfaces in italics.

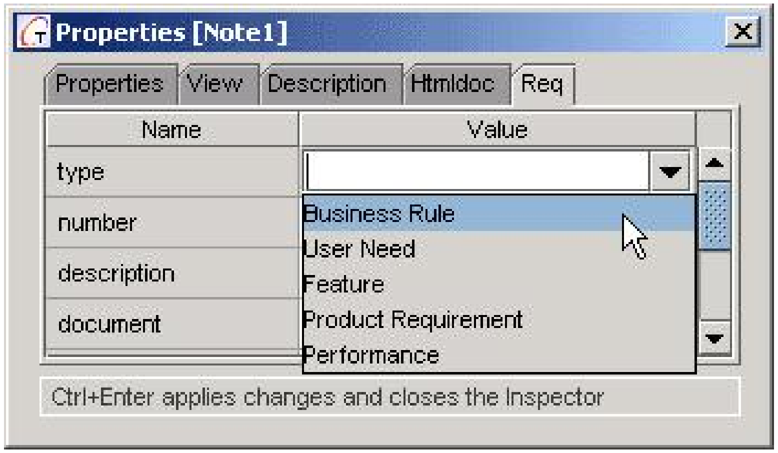
The generalization button also serves for showing that a class implements an interface. (A class implementing an interface must define the operations in the interface. Defer implementing **getName()** in **Agent** and **Reservation** until we discuss patterns.)

Adding notes to diagrams

You're almost finished with a first crack at the airline project. But put in one more item before going on, a note to explain the business rule for making a reservation:

> "You can make a new reservation on a flight if the number of tickets already sold is less than the flight capacity."

Step: Put a note on the **AirlinePD** class diagram to show this business rule. Then set the note type to business rule.



Properties [Note1]

Properties | View | Description | Htmldoc | Req

Name	Value
type	
number	
description	
document	

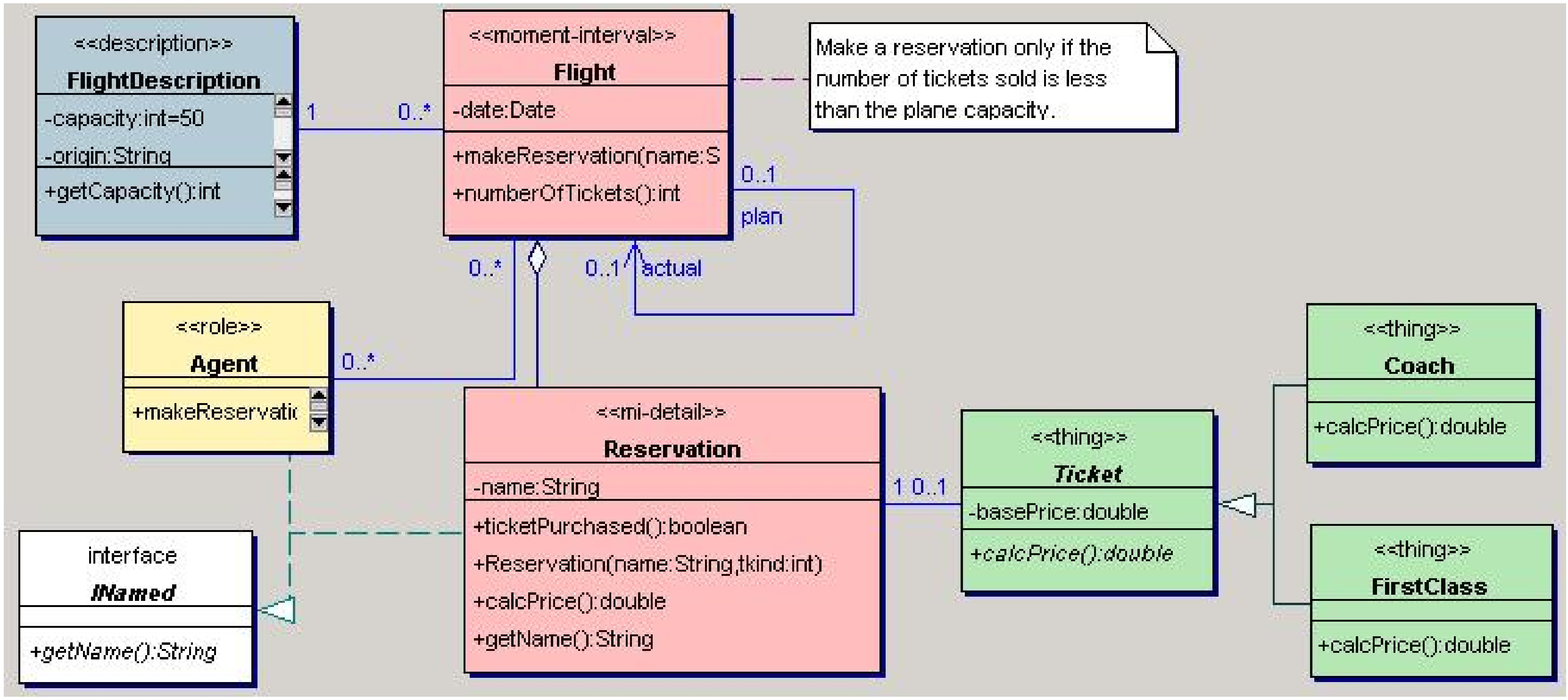
Business Rule
User Need
Feature
Product Requirement
Performance

Ctrl+Enter applies changes and closes the Inspector

Use the note button (▤) on the Diagram pane toolbar -- enter the text directly in the note node on the diagram. When you're finished, link the note to the **Flight** class via the note link button (⤢).

You can set the type of the note on the Req tab of the Properties inspector of the note, as shown here on the right.

Below is a snapshot of our **AirlinePD** class diagram after we created the note. The links are rectilinear. The layout inheritance is horizontal.



Tips and Tricks

- Together will clean up diagram geometry when you select **Layout|All** from the Diagram pane speedmenu.
- If you're unhappy with the any change, whether it's from moving diagram elements, creating or deleting elements, or changing code. You can always use the undo button. There are multiple levels of undo. And if you change your mind in the middle of an undo, click the redo button.
- The undo and redo buttons have flyovers that describe the action that will be undone/redone.

- Having trouble making a self link show on a diagram? Change the diagram layout from rectilinear to direct.
- If you copy an operation from an interface or abstract class to a concrete class, it will be copied as a concrete method. If you copy a method from a class to an interface, it will be copied as a method declaration (with no body).
- If you want an association to appear in a class node, change its name not to begin with **lnk**. Alternatively, select **Options|Diagram** from the Main menu. Then on the diagram tab, select **Associations|Show as attributes|All**.
- To show all links as directed, select **Options|Diagram** from the Main menu and then pick **Associations|Draw Directed|All**.

← PREVIOUS | TOP ↑ | NEXT → | START HOME

Last revised: Thu, Apr 12, 2001

Together Tutorial
Part 6: Template Patterns

Patterns are perhaps the most powerful tool for code reuse today. Patterns can be used to create new classes or members. They can also be applied to existing classes and members.

Together has two different kinds of pattern support: template patterns and "module" patterns. Templates are useful for simple classes as well as links or attributes. Modules are typically used for more complicated design patterns such as GoF design patterns or Coad class patterns.

This part of the Together Tutorial will focus on template patterns. At the end, we will show how to create your own class template pattern.

Contents:

Choosing class member patterns

There are two built-in class member patterns: Stub Implementations and Properties. Stub Implementations put dummy return statements in non-void operations. Properties create attributes with getters and setters.

Step: Apply the Stub Implementation pattern to three methods:

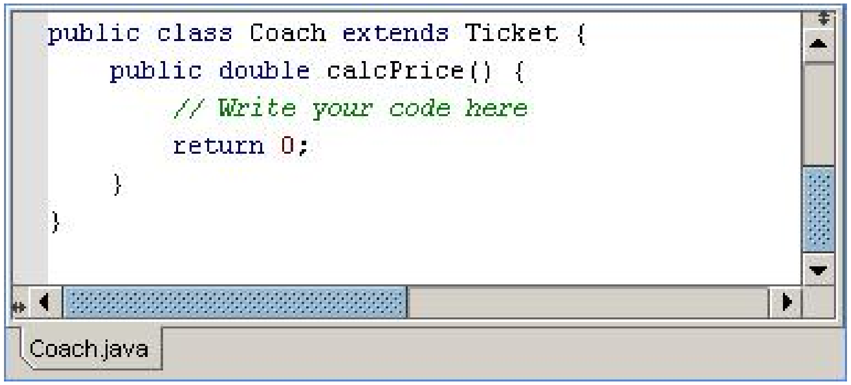
1. **Coach.calcPrice()**
2. **Reservation.ticketPurchased()**
3. **Flight.numberOfTickets()**

Then copy **calcPrice()** from **Coach** to **FirstClass**. (Don't change **Reservation.calcPrice()**. We're saving the implementation of that operation for later.)

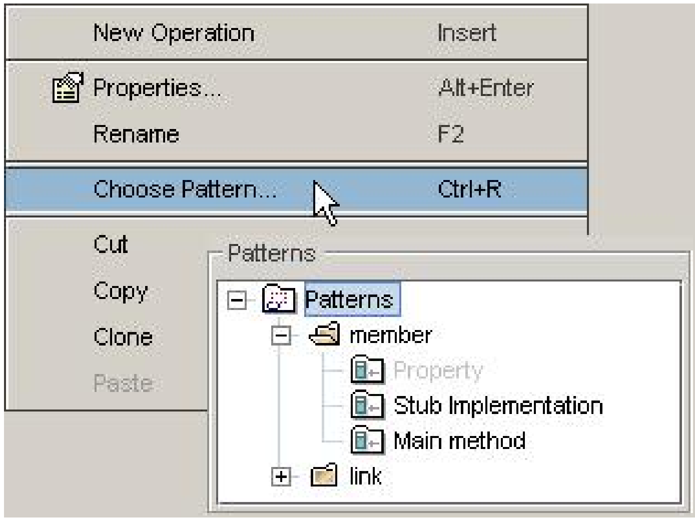
To apply the stub pattern, start with **Choose Pattern** from the operation speedmenu.

Together will open a Pattern window that lists member and link patterns. The only choice for operation patterns is Stub Implementation.

The editor will show the result.

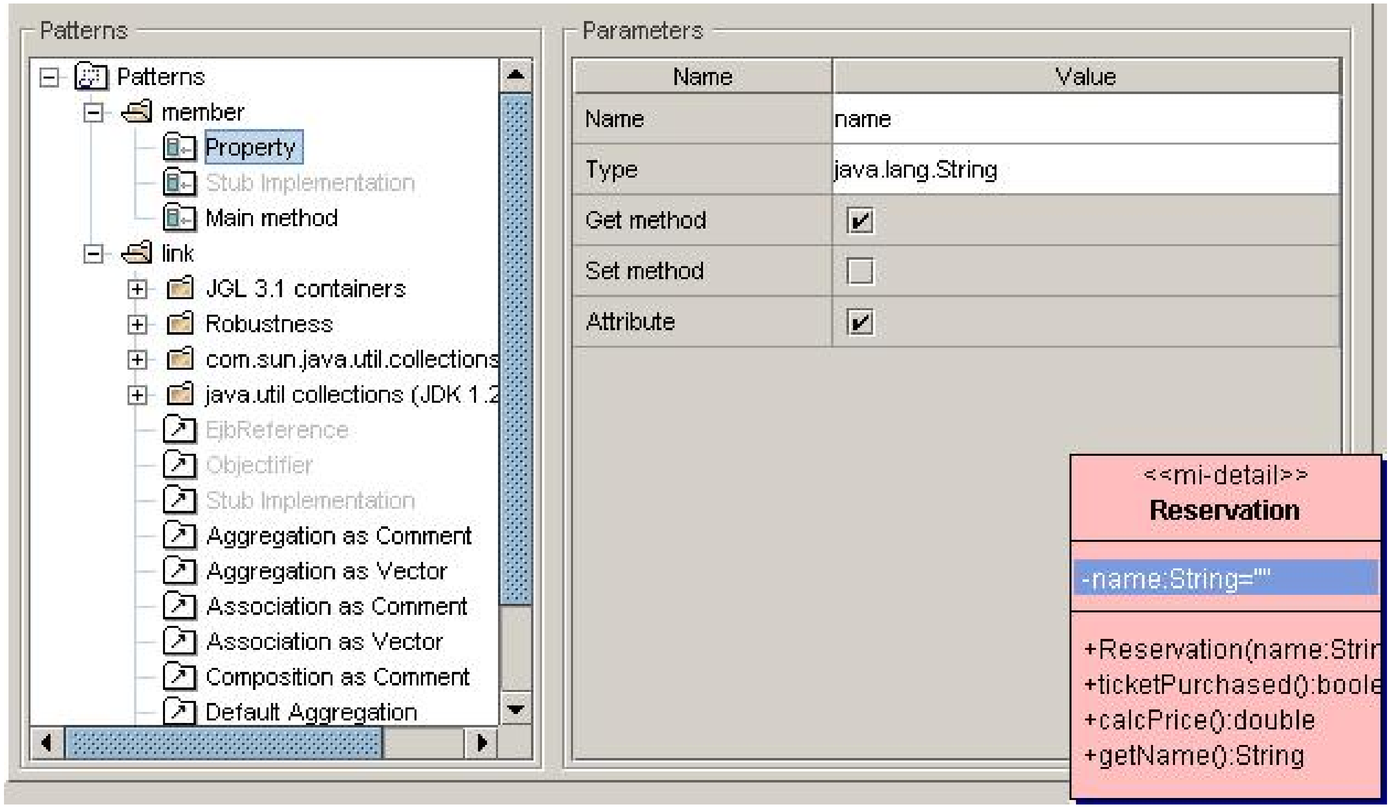


```
public class Coach extends Ticket {
    public double calcPrice() {
        // Write your code here
        return 0;
    }
}
```



Step: Apply the Property pattern to **Reservation.name()** and **Agent.name()**. Give each property a **get** method but not a **set** method.

Again, start with the member speedmenu to apply a pattern. When you select Properties from the list of patterns, the right side of the Pattern window lets you change the Name, the Type, and whether the property has a getter, setter, or attribute.

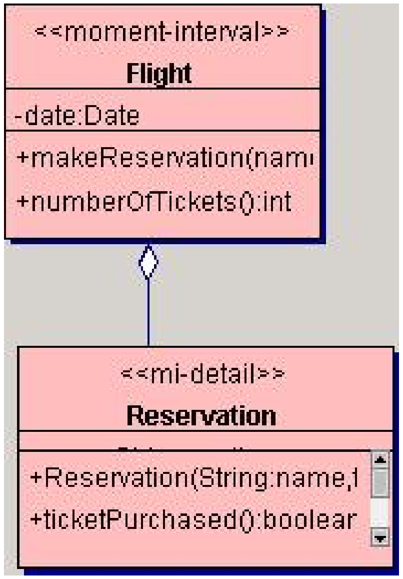


Name	Value
Name	name
Type	java.lang.String
Get method	☑
Set method	☐
Attribute	☑

If you change the text in the Name textfield, Together will change the name of the attribute and methods. If you uncheck the Attribute box, Together will remove the attribute altogether.

If you want to create an new property from scratch rather than use an existing member, use the Member by Pattern option on the class speedmenu.

Applying patterns to links



To the left is the part of the **AirlinePD** class diagram showing the association from **Flight** to **Reservation**. The UML diagram shows only that the association is an aggregation. The diagram reveals nothing about the actual implementation. Is the aggregation coded as a collection, an array, a vector, a hash table?

It is up to the programmer to choose the appropriate code. Link patterns give some help.

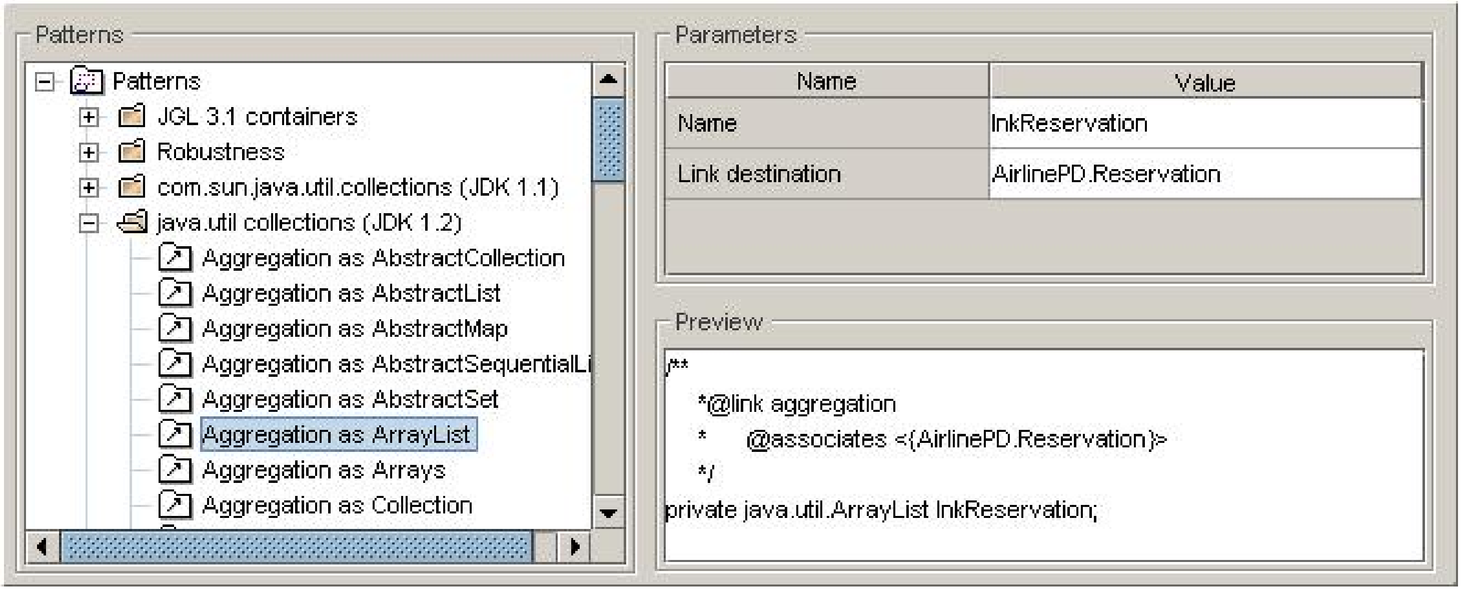
Step: Apply the **Aggregation as ArrayList** pattern to the link from **Flight** to **Reservation**.

There are two ways to use a link pattern:

1. For an existing link, select **Choose Pattern** from the link speedmenu.
2. If the link does not exist, create a link by pattern. The link by pattern button (▨) is on the Diagram toolbar.

In both situations, Together brings up the Association Pattern dialog box.

The Pattern selection pane of the dialog box lists many link patterns. For the last step, select **Association as ArrayList** from **java.util collections**.



Patterns

- Patterns
  - JGL 3.1 containers
  - Robustness
  - com.sun.java.util.collections (JDK 1.1)
  - java.util collections (JDK 1.2)
    - Aggregation as AbstractCollection
    - Aggregation as AbstractList
    - Aggregation as AbstractMap
    - Aggregation as AbstractSequentialLi
    - Aggregation as AbstractSet
    - Aggregation as ArrayList
    - Aggregation as Arrays
    - Aggregation as Collection

Parameters

Name	Value
Name	lnkReservation
Link destination	AirlinePD.Reservation

Preview

```
/**
  *@link aggregation
  *     @associates <{AirlinePD.Reservation}>
  */
private java.util.ArrayList lnkReservation;
```

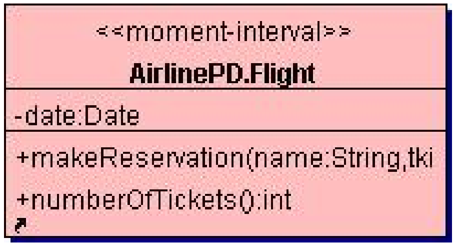
The preview panel on the Association Pattern window indicates the code that Together will generate in the **Flight** class. (The actual code has an import statement rather than fully qualified **ArrayList** type name.)

Applying class template patterns

It's time to create a **Driver** class with a **main** method. The **Driver** class belongs in the **UserInterface** package instead of **AirlinePD**. Before starting the next step, open **UserInterface** in a new tab.

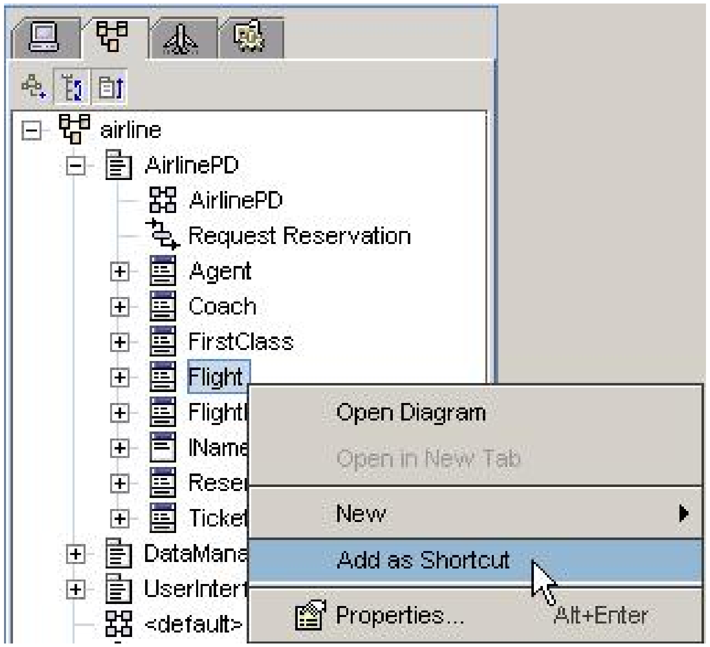
Step: Create a shortcut (alias) to the **Flight** node in the **UserInterface** package diagram.

Start this step by going to the model view of the Explorer. Expand **AirlinePD**, then bring up the **Flight** class speedmenu. Select **Add as Shortcut**.



The **UserInterface** diagram will show a new node, with the shortcut symbol (↗) in the lower left corner.

The Explorer pane shows the shortcut also. Look for a **Flight** class with a shortcut symbol under **UserInterface**.



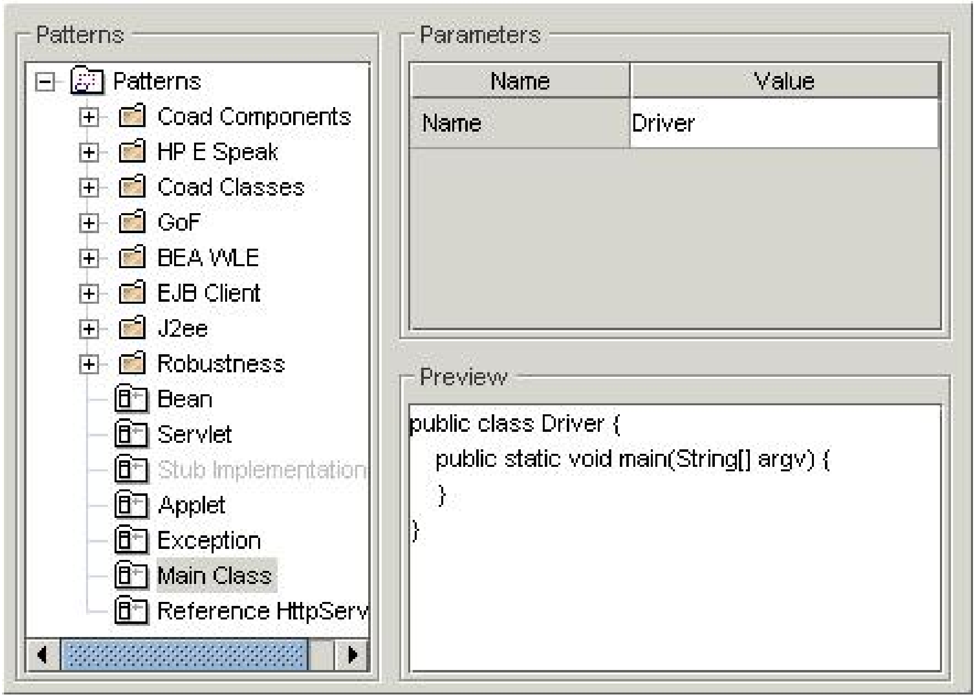
Step: Create a **Driver** class using the **Main Class** template.

The Class by Pattern button (▤) is on the Diagram toolbar. (The three dots at the bottom characterize pattern buttons.)

Placing a Class by Pattern in the Diagram pane brings up a pattern window like the one shown here that we obtained.

We selected the **Main Class** pattern, then changed the name from the default **Class1** to **Driver** in the Name text field on the Parameters pane.

The Preview pane shows the code that Together generates.



Name	Value
Name	Driver

```
public class Driver {
   public static void main(String[] argv) {
   }
}
```

While the **Main Class** pattern may not seem like much of a time saver, some of the other choices might be more impressive.. For example, take a look at the **Bean**, **Applet**, and **Servlet** class patterns.

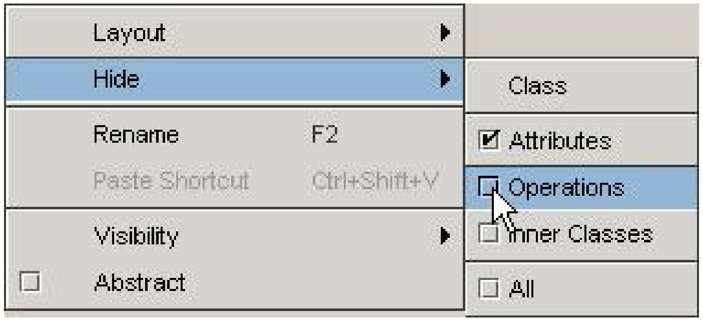
Step: Make a link from the **Driver** node to the **AirlinePD.Flight** node. Then rename the link as **myFlight**.

That last step has nothing to do with patterns, but something to do with views.

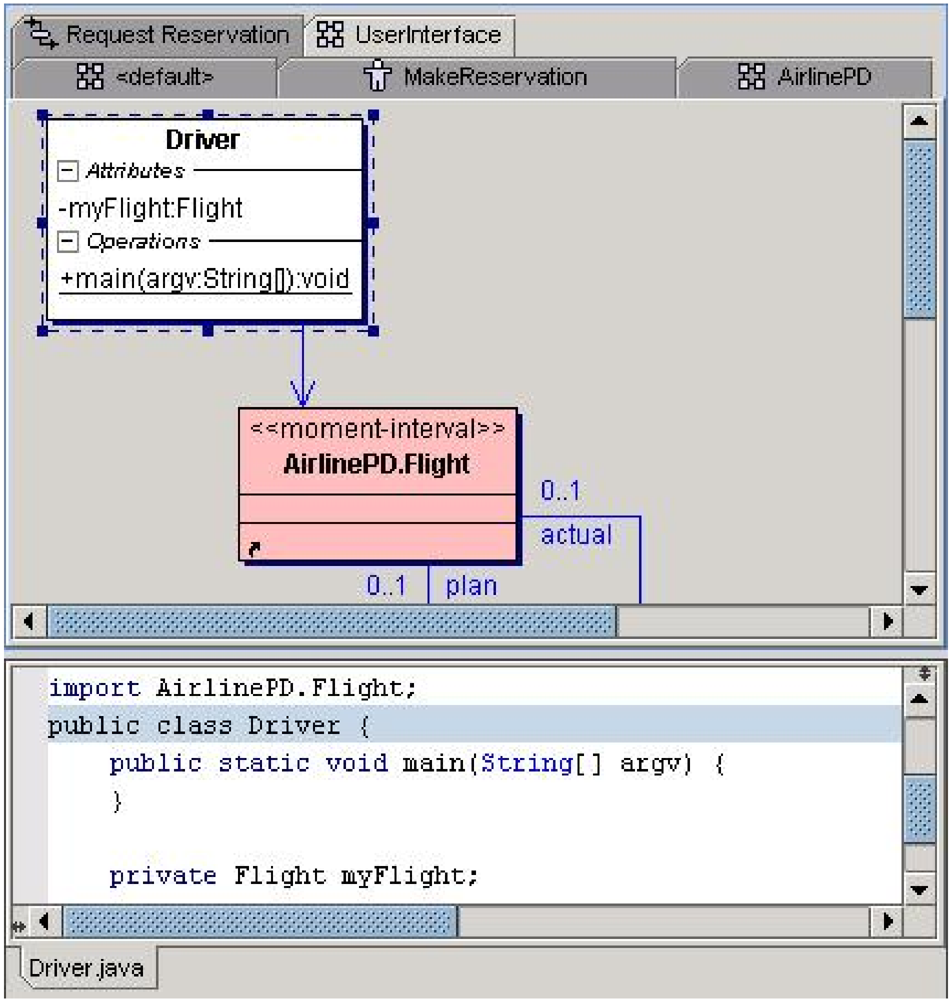
To the right are the Diagram and Editor panes that we generated. There are two things to notice:

1. When we created the link, Together put the appropriate import statement in our code.
2. The **Flight** node does not show any of its members.

You can hide class members on a diagram node via the node speedmenu.



For the **Flight** node, checking **Attributes** and **Operations** is equivalent to checking **All**.



```
import AirlinePD.Flight;
public class Driver {
    public static void main(String[] argv) {
    }

    private Flight myFlight;
```

Making your own class template pattern

Java class templates reside in the Together home folder under **templates/JAVA/CLASS**. Each folder in that directory contains two files:

1. **%Name%.java**, which is the template for the source
2. **FolderName.properties**, which establishes the properties for creating code and documentation

Together 5.0 has a Code Template Expert for creating new templates or modifying existing ones. The Expert will automatically create a folder for a new template, placing inside the two files listed above.

Step: Create an **Exception** template through the template expert.

Invoke the Code Template Wizard under the Main menu. Select **Tools|Code Template Expert**.

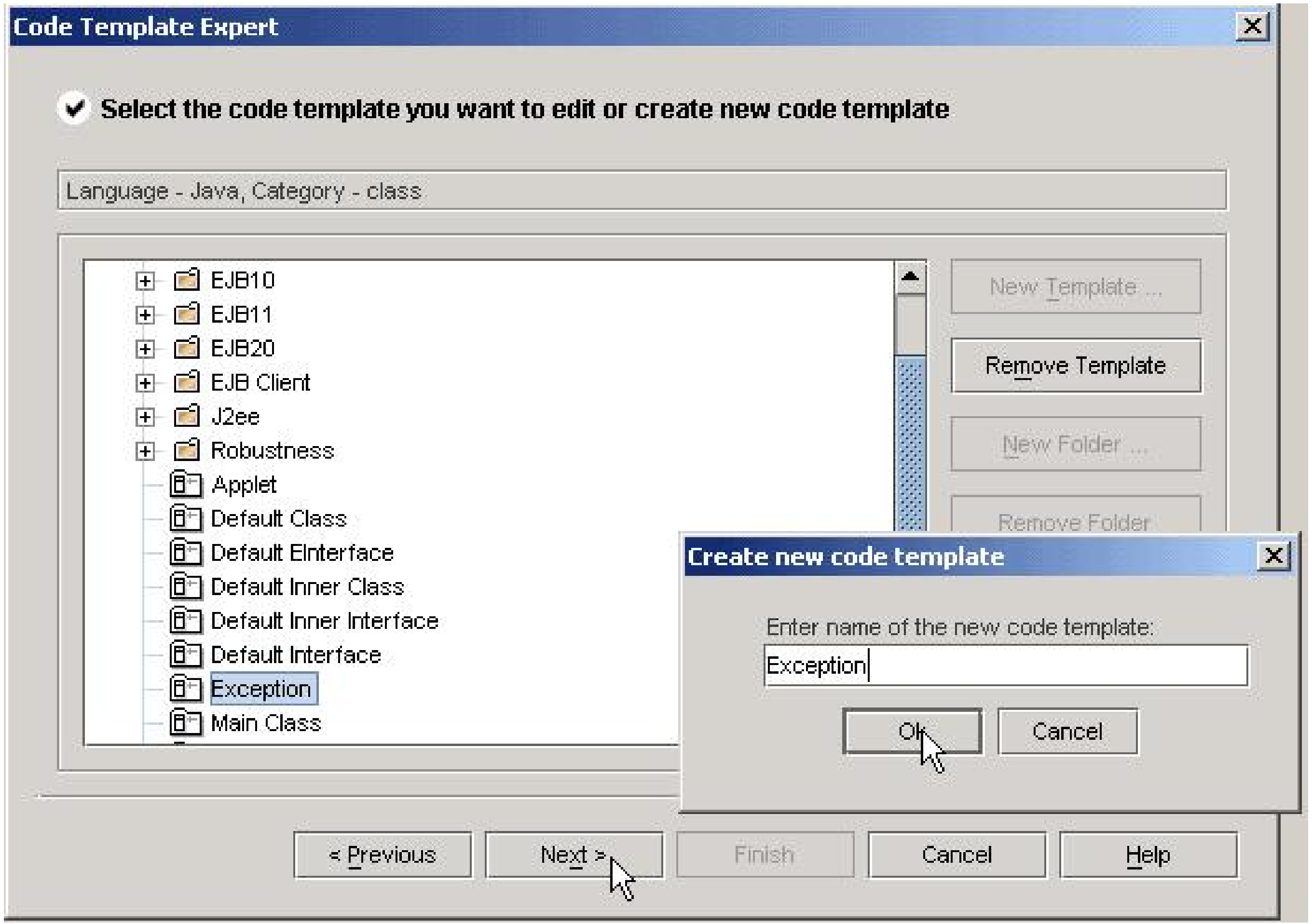
In the first two panes of the Code Template Expert, select the Language, Category, and New Template.

1. Pane 1: **Select language and category of desired code template.**
   Select as follows.
   Template Language: Java
   Template Category: Class

2. Pane 2: **Select the code template you want to edit or create new code template.**
   Click **New Template**.

Enter **Exception** in the dialog box, then click **OK**. Together puts you back at the same pane.

Choose **Exception** from the scrolling list. Then click **Next** to continue to Pane 3.



Code Template Expert

Select the code template you want to edit or create new code template

Language - Java, Category - class

Create new code template

Enter name of the new code template:

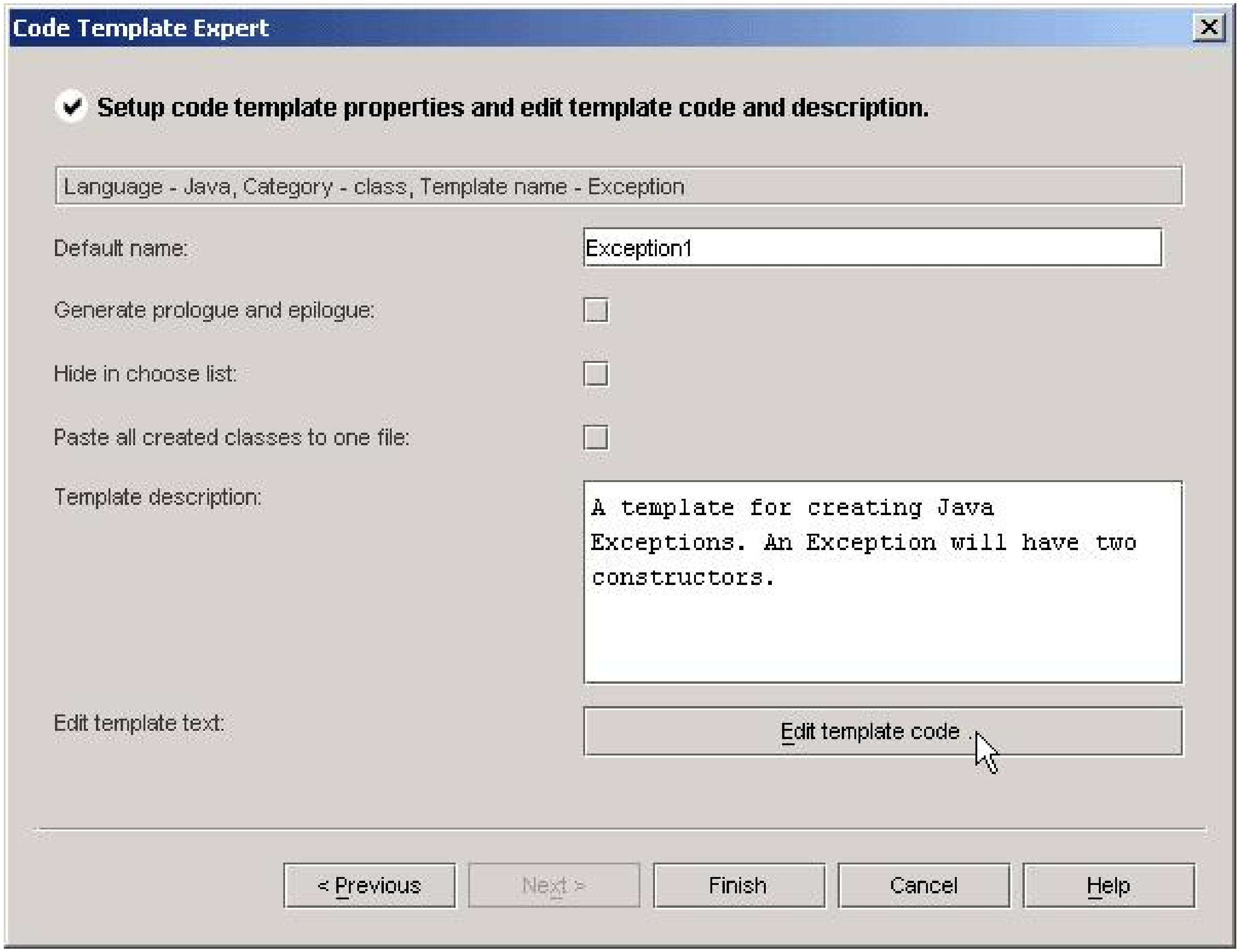
Exception

Step: Fill in the Exception details, beginning with the template documentation. The default name should be Exception1. Create an appropriate description for the dialog box.

The snapshot at the right shows our template so far.

1. **Default name:** Exception1
   Unless we enter a different name, the name of any new class generated by the template will be Exception1.

2. Template description is documentation for the template dialog box when the Exception template is selected. (The description is in HTML.)



Code Template Expert

Setup code template properties and edit template code and description.

Language - Java, Category - class, Template name - Exception

Default name: Exception1

Generate prologue and epilogue: □

Hide in choose list: □

Paste all created classes to one file: □

Template description:

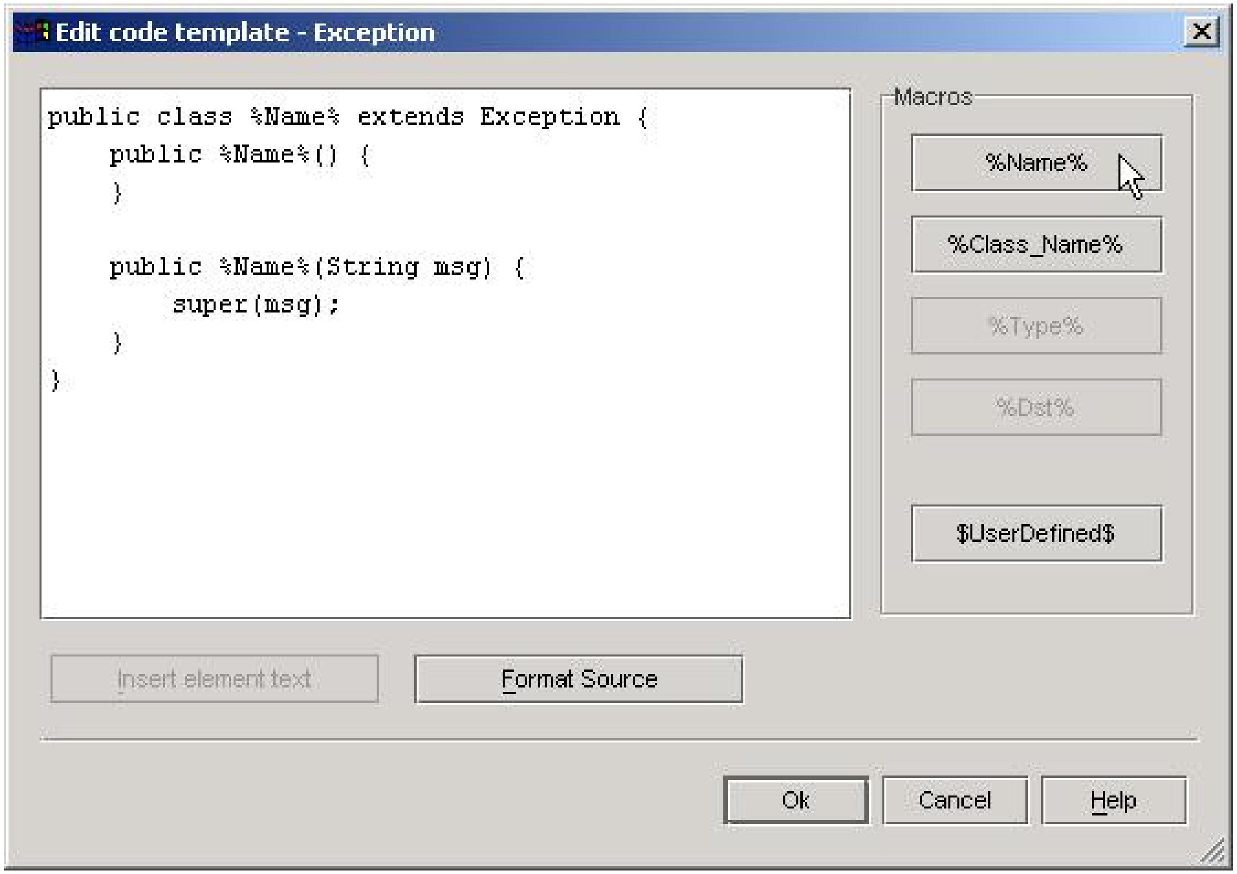
```
A template for creating Java
Exceptions. An Exception will have two
constructors.
```

Edit template text:

Step: Fill in the template definition, and finish the template.

Pane 3 of the Code Template Expert has a window for filling in the code.

1. Fill in the definition as illustrated here. You can click on the **%Name%** button rather than typing in the 6 characters. (The text in the Name field will replace the %Name% macro in the definition when the template is used.)
2. Click **Format Source** after entering the source code.
3. Click **OK** when you are finished with the code.
4. Click **Finish** on the following pane to complete your work.



```
public class %Name% extends Exception {
    public %Name%() {
    }

    public %Name%(String msg) {
        super(msg);
    }
}
```

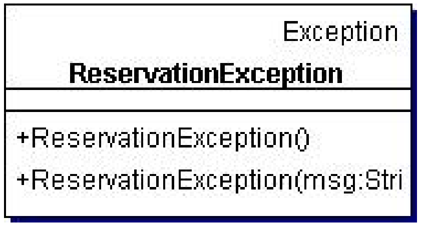
Step: Create an **Exception** class in the **AirlinePD** package named **ReservationException**.

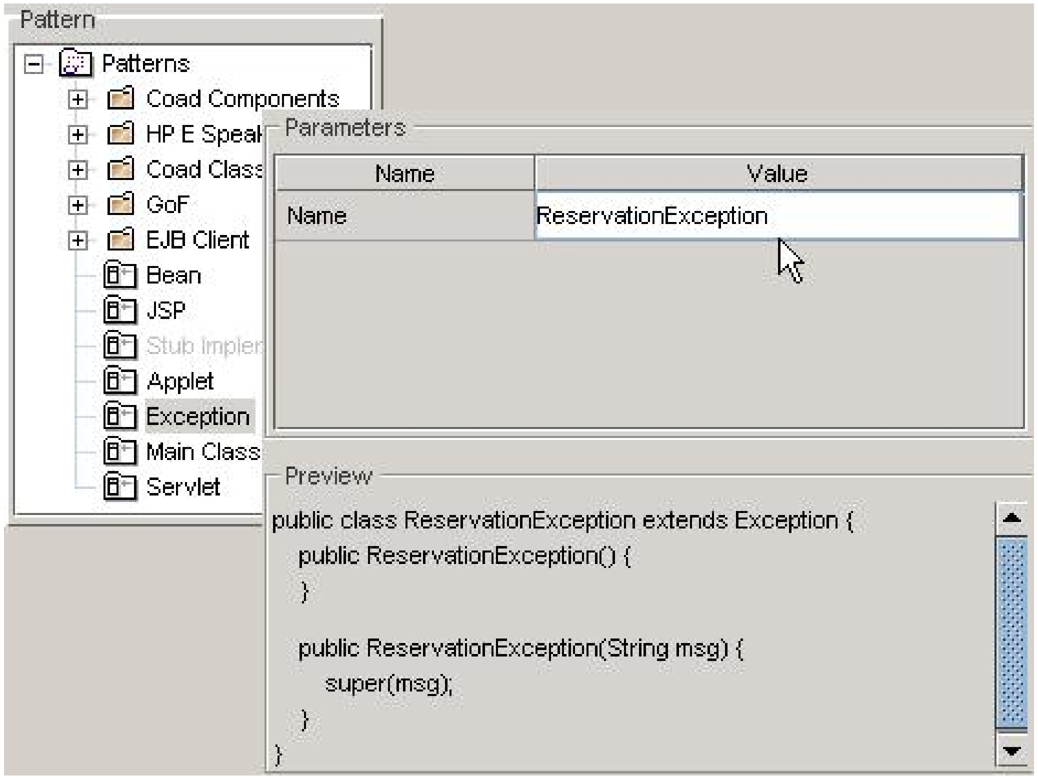
Go back to the **AirlinePD** package diagram to start this step.

When you click the Class by Pattern button (▤) on the diagram toolbar then the diagram, the pattern dialog box should show your editing work. Select **Exception** as the pattern. The Description panel has the HTML description from the properties file.

Enter the name **ReservationException** in the textfield of Parameters panel. The Preview panel will show the code that Together will generate as a result.

When you finish, the new **ReservationException** node will appear on the diagram.





Name	Value
Name	ReservationException

```
public class ReservationException extends Exception {
  public ReservationException() {
  }

  public ReservationException(String msg) {
    super(msg);
  }
}
```

(We suggested creating an exception for this final set of exercises in order to satisfy the business rule described in the activity diagram. When a request for a flight reservation is refused because the plane is full, **Flight.makeReservation()** can throw a **ReservationException**.)

Tips and Tricks

- Using the Diagram toolbar to create a class from a template is almost identical to using a speedmenu on an existing class or interface. With the speedmenu, Together is often able to fill in a name from your existing code. (Using the Diagram toolbar is absolutely identical to using the Diagram speedmenu to create a class, interface, package, or class by pattern.)
- Shortcuts are aliases for the real thing. You can delete a shortcut from a diagram if you select it and press the Delete key. If you use delete on the shortcut speedmenu instead, Together will remove the corresponding file (not just the shortcut).
- It is as easy to make new link templates -- or modify existing link templates -- as it is to create new class templates. The Code Template Expert will step you through! Just select Link instead of Class on the Expert's first pane.

← PREVIOUS | TOP ↑ | NEXT → | START HOME

Last Revised: Fri, Mar 30, 2001

Together Tutorial
Part 7: Refactoring with Class Patterns

In the previous part of the Together Tutorial, we discussed template patterns. It's time to consider the more complicated module patterns. The focus will be on the Abstract Factory pattern, which is one of the 11 GoF patterns shipped with Together.

Together patterns are useful for automatic creation of code that is tedious to write. And they are also critical for refactoring code by reorganizing it. That is exactly how you will use patterns in this section. But first, you'll start out with some unfinished business, coding our single business rule.

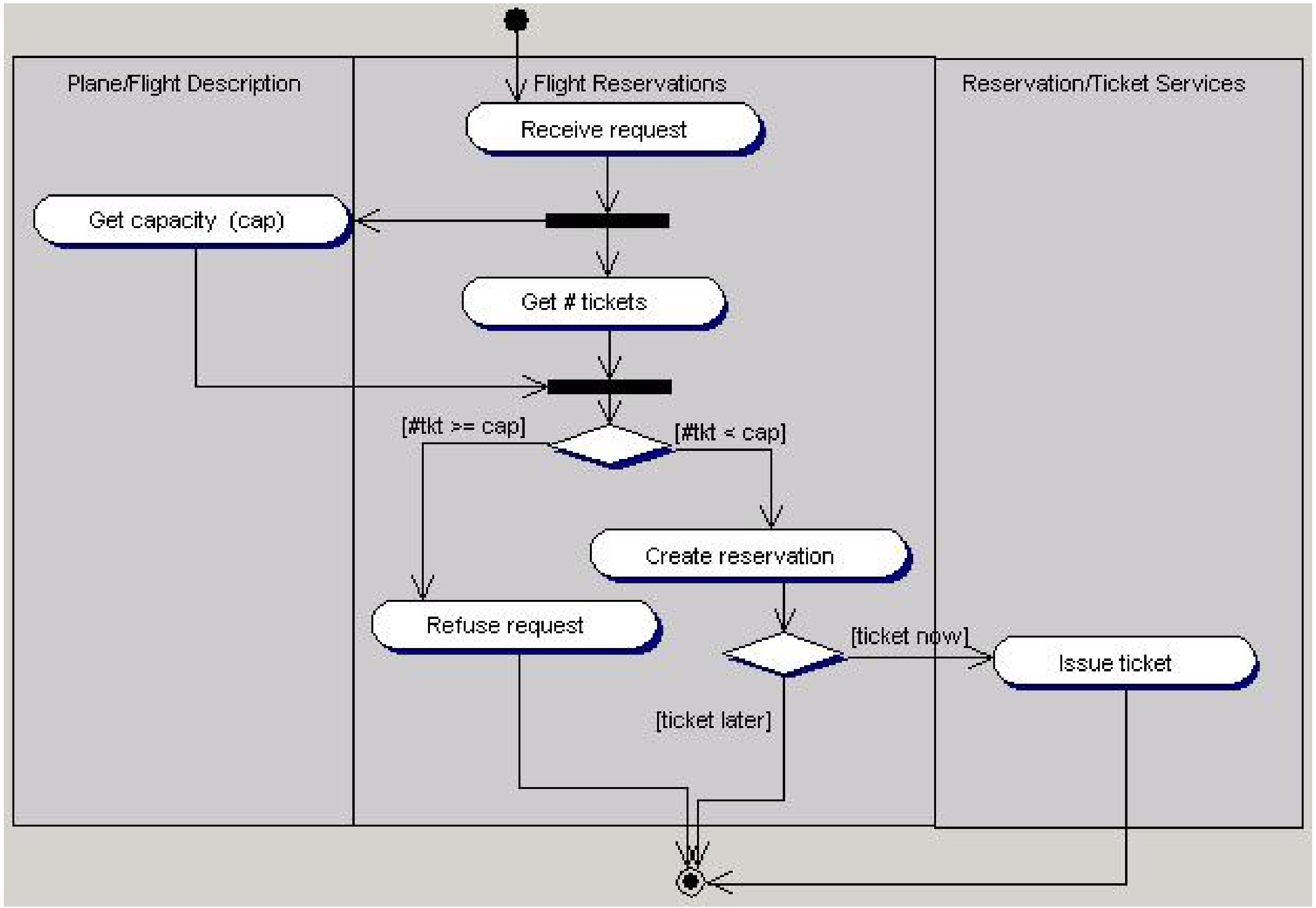
Contents:

GoF is an acronym for "Gang of Four." It refers to Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides, who are the authors of the classic book *Design Patterns, Elements of Reusable Object-Oriented Software.*

Coding business rules

Let's write some simple code to generate a sequence diagram from **Flight.makeReservation()**. Our previous discussion on activity diagrams contains the business rule that describes the code:

> **You can make a flight reservation if the number of tickets sold so far for the flight does not exceed the capacity of the airplane.**



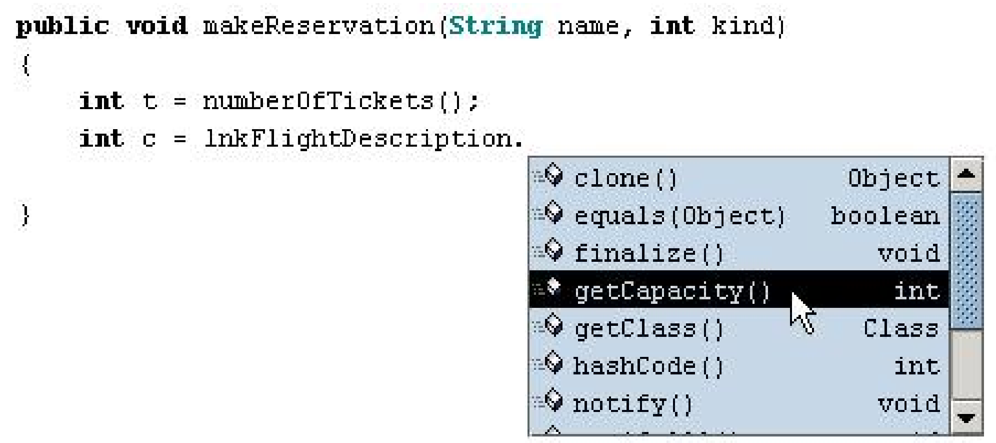
Below is our code for **Flight.makeReservation()**.

```
public void makeReservation(String name, int kind) throws ReservationException {
   int t = this.numberOfTickets();
   int c = lnkFlightDescription.getCapacity();

   // Make a reservation only if the number of tickets sold is less than the plane capacity
   if (t < c) {
      Reservation r = new Reservation(name,kind);
      lnkReservation.add(r);
   }
   else throw new ReservationException();
}
```

Step: Complete the coding of **Flight.makeReservation()**.

You can do this step by copying and pasting from browser to Together. However, if you enter the code for **makeReservation()** by hand, you're likely to see the Together editor code completion at work.

This snapshot shows code completion for the expression **lnkFlightDescription**.



```
public void makeReservation(String name, int kind)
{
    int t = numberOfTickets();
    int c = lnkFlightDescription.

}
```

The **Reservation** constructor is still just a stub. Let's use its two parameters, **name** and **tkind**. The **name** is easy. But **tkind** takes some thought. For now, simply go by this:

> **If tkind is 1, create a Coach class ticket. If tkind is 2, create a FirstClass ticket. Otherwise, do not create a ticket at all.**

Step: Fill in the code for the **Reservation** constructor.

Here is the code for our **Reservation** constructor.

```
public Reservation(String name, int tkind) {
   this.name = name;

  // We will refactor this next piece of code
   lnkTicket = null;
   if (tkind == 1)
       lnkTicket = new Coach() ;
   else if (tkind == 2)
      lnkTicket = new FirstClass();
 }
```

Refactoring with a GoF pattern

Look ahead now to a time when the airline project might be much more extensive. The code for determining which kinds of tickets are purchased lies completely within the **Reservation** constructor. A natural improvement is to factor out that ticket creation into a new type of object, a factory that creates tickets.

In this section, you will make a **TicketMaster** class for creating actual **Ticket**s. The Abstract Factory pattern from the GoF collection will do the hard work.

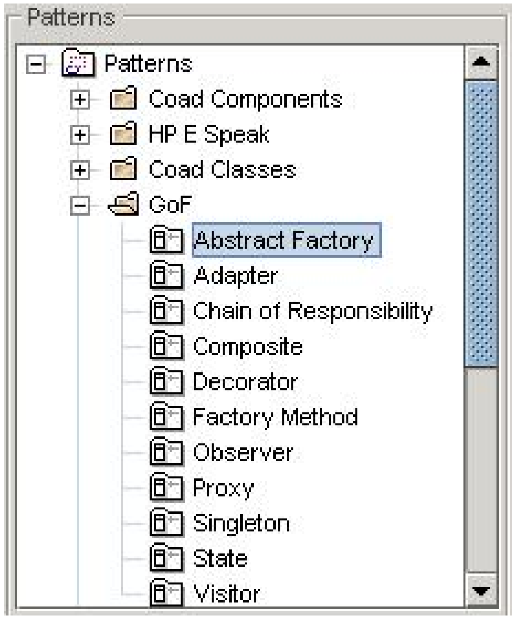
Step: Bring up the dialog for class by pattern using the Abstract Factory from the GoF collection.

Select the Class by Pattern button ( ) on the Diagram toolbar then click on the Diagram pane. That brings up a dialog box with a pattern-picker pane on its left. Expand the GoF folder to see the 11 pattern choices.

Abstract Factory is the first GoF pattern listed. Together forces you to specify the details of the Abstract Factory before it will create new code. As you click on Abstract Factory to select it, Together displays a red warning message at the bottom of the pane:

Product (AbstractProduct) for Factory (ConcreteFactory) is not defined.

It's time to name the Abstract Factory details.



Patterns

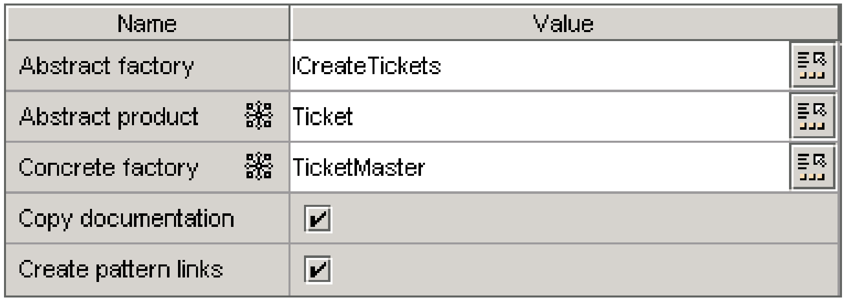
- Patterns
  - Coad Components
  - HP E Speak
  - Coad Classes
  - GoF
    - Abstract Factory
    - Adapter
    - Chain of Responsibility
    - Composite
    - Decorator
    - Factory Method
    - Observer
    - Proxy
    - Singleton
    - State
    - Visitor

Step: Name **Abstract Factory**, **Concrete Factory**, and **Abstract Product** as follows:

- **ICreateTickets** for **Abstract Factory**
- **Ticket** for **Abstract Product**
- **TicketMaster** for **Concrete Factory**

Fill the names for the pattern details in the upper right pane of the dialog box. You'll have to type in **ICreateTickets** and **TicketMaster** directly since they do not yet exist.

You can type in the name **Ticket**, or you can opt to use the select box at the right of the **Abstract Factory** text field instead ( ).



Name	Value
Abstract factory	ICreateTickets
Abstract product	Ticket
Concrete factory	TicketMaster
Copy documentation	☑
Create pattern links	☑

At this point, the pattern is not complete. There's still an error message at the bottom of the pane:

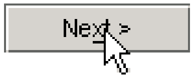
Product (Ticket) for Factory (TicketMaster) is not defined.

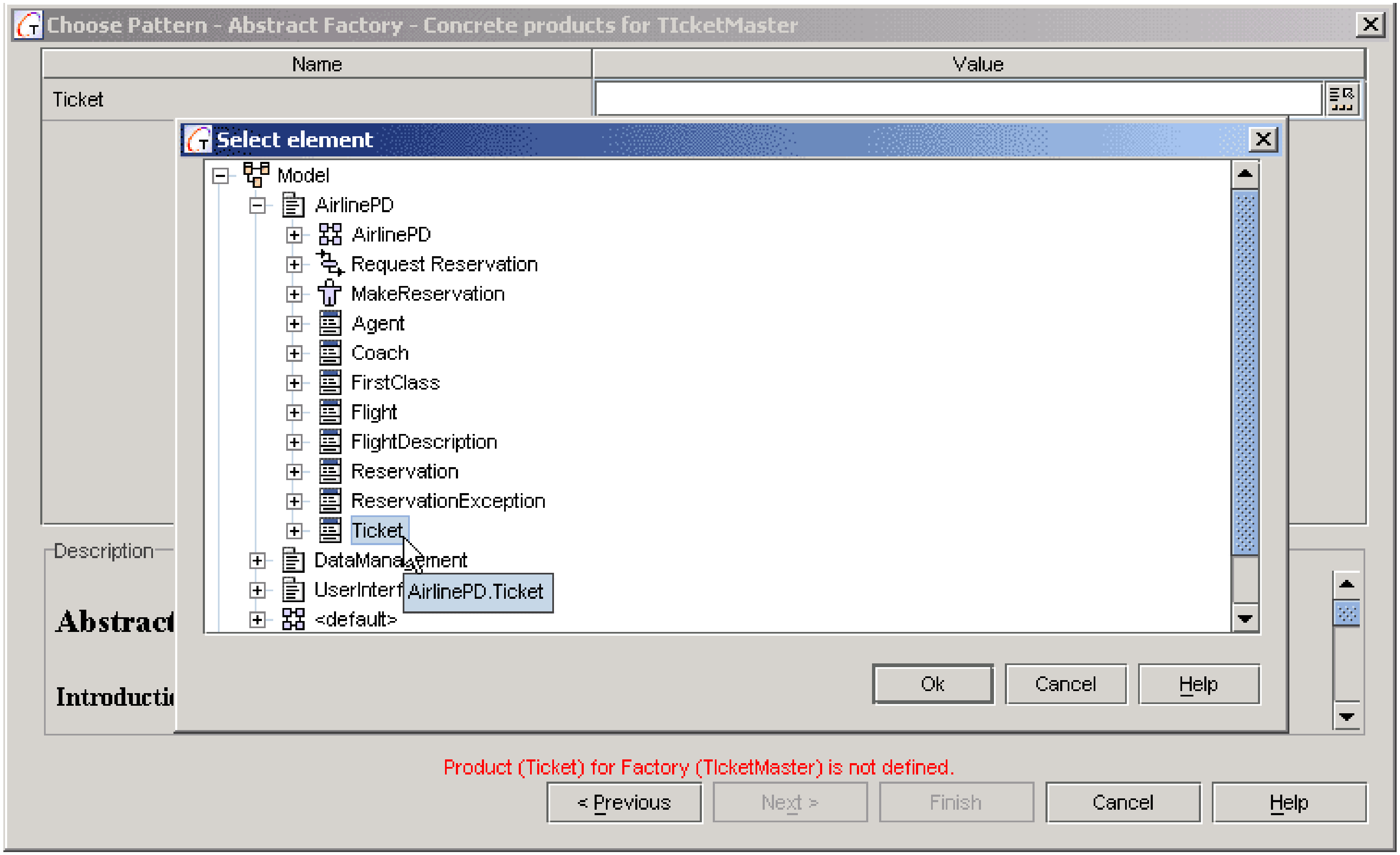
Step: Designate **Ticket** for the **Product** and finish the pattern.

Click the Next button at the bottom of the Choose Pattern window to begin this step. The next pattern window has a text field for entering the product.

Clicking on the select box in that window brings up a Select element window like the one shown below.

To choose the product, expand **Model**, then **AirlinePD**. Then select the **Ticket** class, and click the OK button.





Product (Ticket) for Factory (TIcketMaster) is not defined.

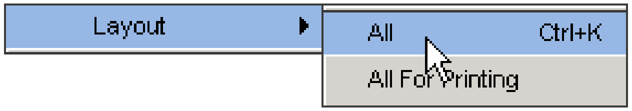
At this point, there should be no red error messages. Click the Finish button to complete creating the pattern.

Putting the finishing touches on diagram and code

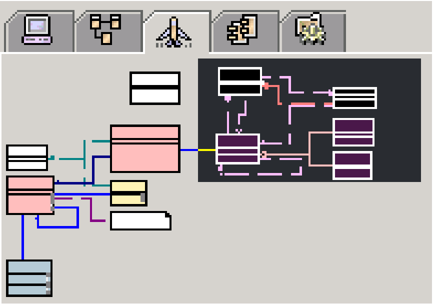
Creating a pattern can result in a pretty messy diagram, especially if new classes or interfaces are generated in the process. You now have a bunch of new dependencies plus the **TicketMaster** class and the **ICreateTickets** interface.

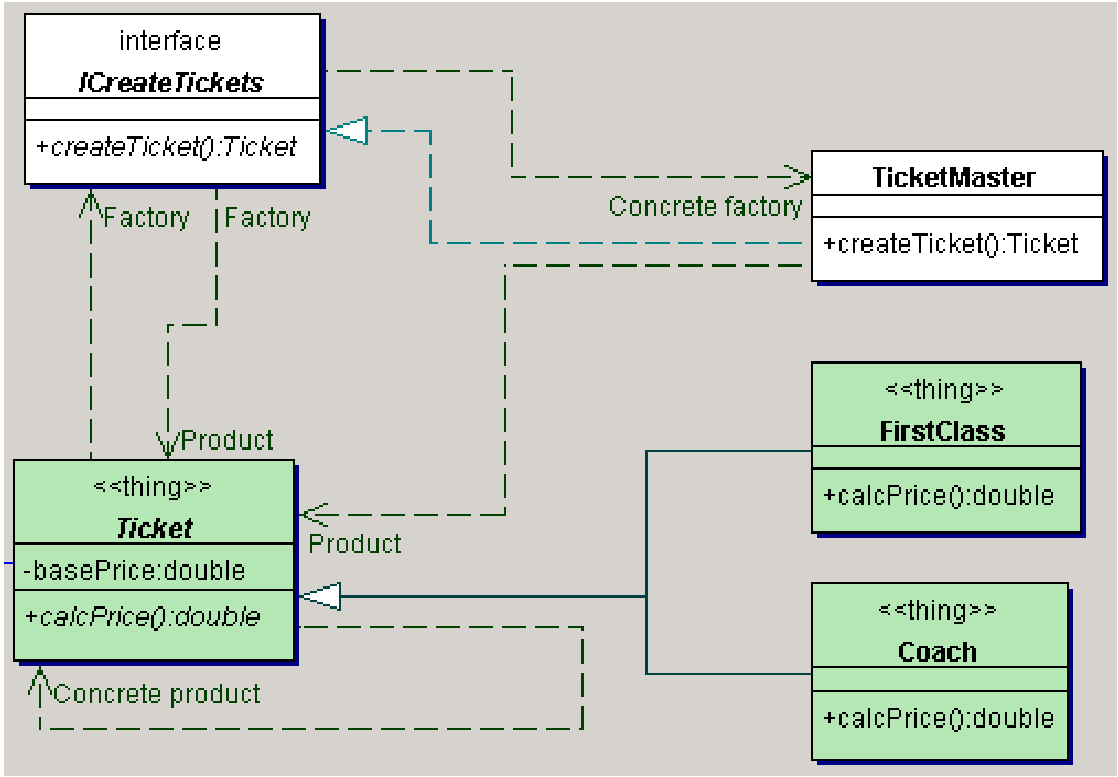
Step: Rearrange the **AirlinePD** diagram to show the overall model shape.

This is an easy step. Bring up the diagram speedmenu and select **Layout|All**.



Below is our snapshot from the Overview tab of the **Explorer** pane. The corresponding **Diagram** pane is at the right. (Our links are rectilinear.)





At this point, the code is almost complete, but not quite. Remember that we originally wanted to factor the code for creating a **Ticket** out of the **Reservation** class and into **TicketMaster**. That describes your next steps.

Step: Change the **ICreateTickets** operation to take an **int** parameter and make the corresponding change in **TicketMaster**.

This takes two easy mini-steps:

1. Change the operation in **ICreateTickets** by using the in-place editor:

   createTicket(tkind:int):Ticket

2. Ctrl+drag the operation from **ICreateTickets** to **TicketMaster**. Delete the original **TicketMaster.createTicket()** that has no parameter.

Step: Move the **Ticket** creating code from **Reservation** to **TicketMaster** and correct the syntax.

You can move the original code by cutting and pasting in the **Editor** pane. But the code won't compile immediately after that. You'll need a couple of minor changes -- declare a **Ticket** variable at the beginning and return it at the end.

We renamed our **Ticket** variable **t** (rather than **lnkTicket**), as shown below.

```
public Ticket createTicket(int tkind) {
   Ticket t = null;
   if (tkind == 1)
       t = new Coach() ;
   else if (tkind == 2)
      t = new FirstClass();
   return t;
 }
```

It would be a fine idea to document those magic numbers 1 and 2 (or better yet, get rid of them altogether). But we'll leave that to your discretion.

Step: Complete the **Reservation** constructor so that it uses a **TicketMaster** to create the appropriate kind of **Ticket**.

Here is our new, improved **Reservation** constructor.

```
public Reservation(String name, int tkind) {
   this.name = name;

   TicketMaster tm = new TicketMaster();
   lnkTicket = tm.createTicket(tkind);
 }
```

Step: Check your work by compiling your code. Fix all syntax errors. (They should be minor if any.)

Make and build commands are on the **Main** menu under **Tools** as well as on the main toolbar. You can also find them on the Builder tab of the Message pane. Compiler output messages are on the Message pane.

If you have errors, you can click on the error message to highlight the bad code in the Editor pane. Be sure to fix your code before continuing the tutorial.

Tips and Tricks

- There are other GoF patterns that are appropriate for this project. For example, the Composite pattern gives be an easy way to allow for group or individual reservations.
- Select several classes at a time in the Diagram pane by left-click-drawing a rectangle that touches all of them. Once they are selected, you can move them as a group.
- Ctrl+drag to copy an operation from an abstract class or interface to a concrete class. Together makes the resulting operation concrete.
- Use the Diagram pane to navigate in the Editor. For example, if you want to copy statements from one method to another, navigate to the source by clicking its method in the Diagram. Then copy the code with the usual commands. Then navigate to the target method via the Diagram pane and copy.

← PREVIOUS | TOP ↑ | NEXT → | START HOME

Last Revised: Mon, Apr 9, 2001

Together Tutorial
Part 8: Sequence Diagrams

The class diagram gives the overall shape to a model. But it's a static diagram, independent of activity within the model. In this part of the Together Tutorial, you will create some sequence diagrams to show how activities occur.

Together can generate sequence diagrams from actual code. And it will generate code from sequence diagrams, including:

- Class declarations.
- Operation declarations.
- Statements in methods that do not yet contain any statements. (Done on explicit request only.)

This part of the tutorial will cover generating code and sequence diagrams. At the end is a discussion on how to create hyperlinks to tie together related project entities.

Contents:

Creating a sequence diagram scratch pad

Begin your work by bringing the **AirlinePD** diagram into focus. The first sequence diagram will be for activity in the problem domain.

Step: From the **AirlinePD** package, create a sequence diagram named **FindRevenue**.

To create a new diagram, click the left most button on the Main toolbar (▣). Below is a snapshot of our new diagram dialog. We filled in a diagram description as well as name.

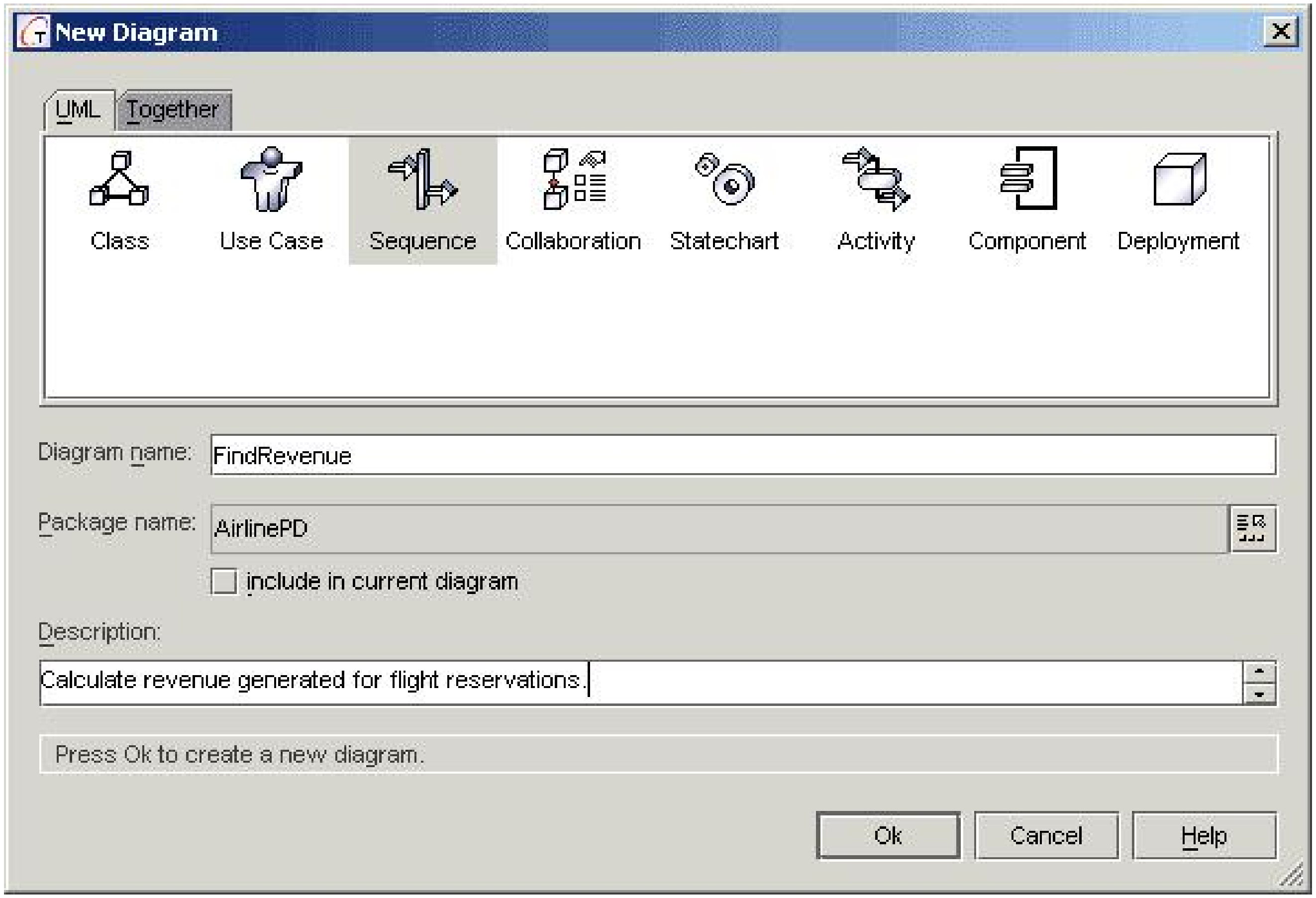


Diagram name: FindRevenue

Package name: AirlinePD

□ include in current diagram

Description:

Calculate revenue generated for flight reservations.

Press Ok to create a new diagram.

The sequence diagram toolbar has five items of interest for the diagrams on this page.

	An actor	can start the message chain
	An object	receives and sends messages
➡	● A message	link between objects
↵	● A self message	call to a method on the same object
{ }	● A statement block	for control statements (loops, etc.)

Step: Place an actor on the left side of the sequence diagram. Then put in three objects.

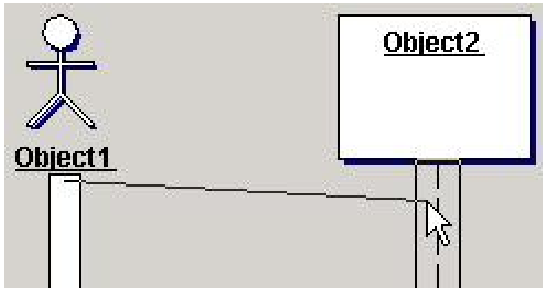
Together draws a dotted lifeline below each object except the actor, which has a narrow rectangle instead.

Step: Create a message from the actor to the **Object2** lifeline.

You can drag a message from one lifeline to another. The action is similar to dragging an association from one object to another on a class diagram and dragging a transition for activity diagrams.

When you finish this step, the target lifeline will have an activation bar (rectangle) starting at the point of the arrow. (An activation bar can be lengthened, shortened, and moved along the lifeline.)

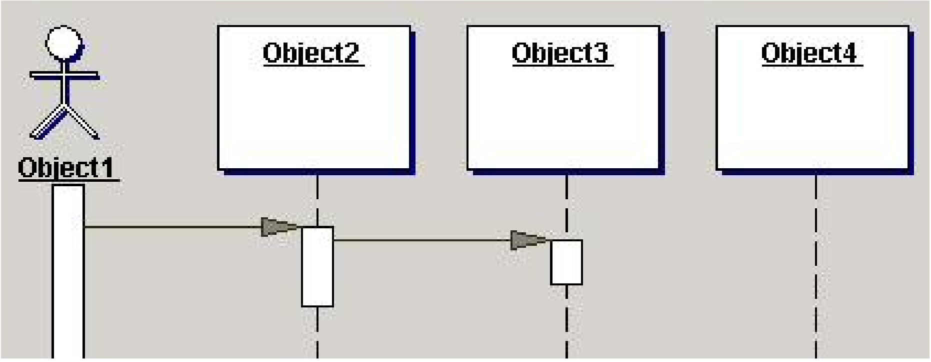
Together puts a halo around the valid source and target lifelines as you pass the cursor over them.



Step: Create a message from the activation bar on **Object2** to the **Object3** lifeline.

Make sure that you start that last message within the activation bar and not below. Otherwise, you will get a new, separate activation bar on the **Object2** lifeline.

Our sequence diagram snapshot shows strictly generic classes and messages, completely unrelated to classes or operations in the class diagram.



Correlating generic objects with classes and generic messages with operations

You can convert generic objects to instances of existing or new classes. And you can make the generic messages correspond to actual operations on these classes.

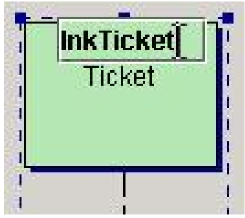
Step: Going left-to-right, select classes for each object (except the actor):

- for **Object2**, choose **Flight**
- for **Object3**, choose **Reservation**
- for **Object4**, choose **Ticket**

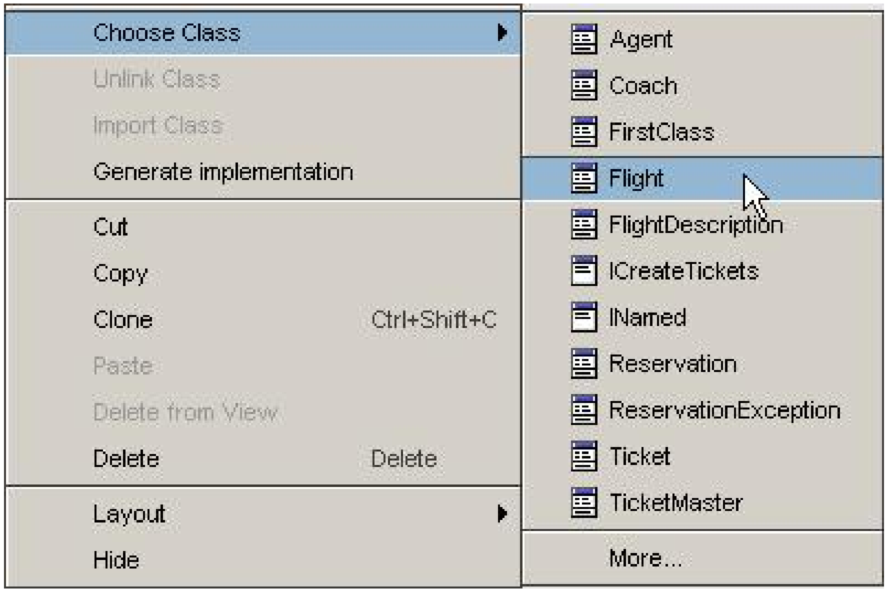
Then rename the right most object (from **Object4** to **lnkTicket**).

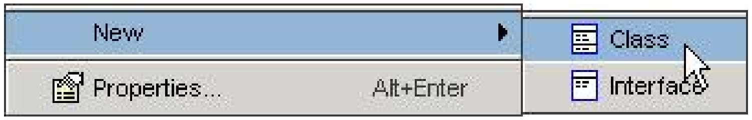
Begin this step by using the object speedmenus. On the right is the speedmenu for **Object2**. **Choose Class** lists the classes from the package of the sequence diagram. (**More** at the bottom of the choices offers classes outside the package.)

After you pick the classes, your sequence diagram should show the three objects and their activation bars in the pink and green stereotype colors.



The easiest way to change the name of an object through the in-place editor.



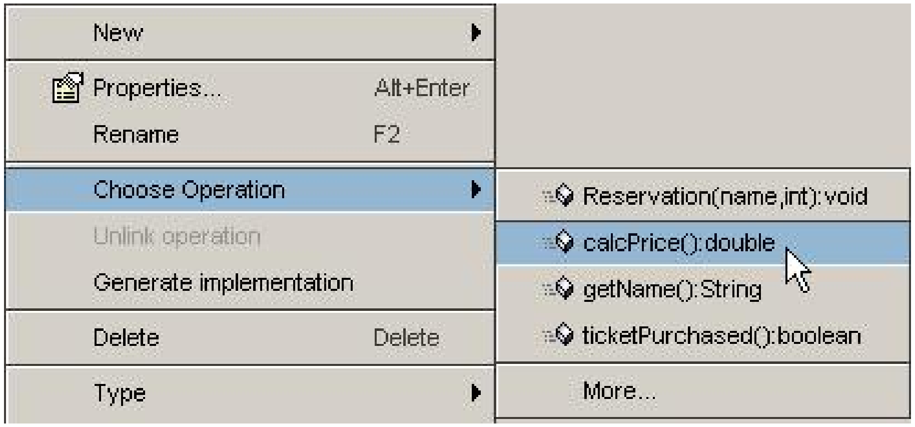


Incidentally, the object speedmenu gives the option of creating a new class or interface. If you select one of those, Together will generate the corresponding code (and change any corresponding class or package diagram).

Step: Choose **calcPrice()** for the message from **Flight** to **Reservation**.

Right clicking on the message in the diagram brings up its speedmenu.

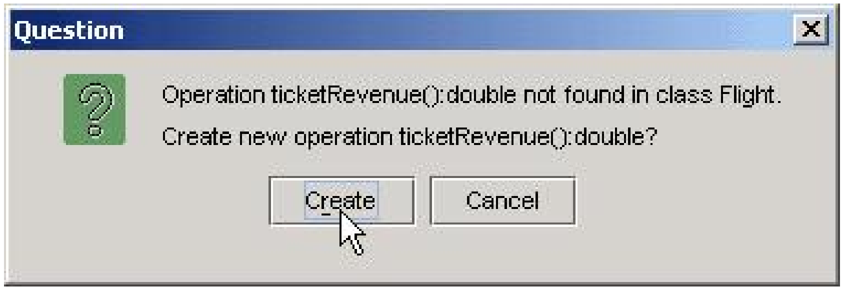
Choose Operation on the speedmenu gives a list of the available class methods.



Step: Create a new operation for the generic message from the actor to the **Flight** object -- **ticketRevenue():double**.

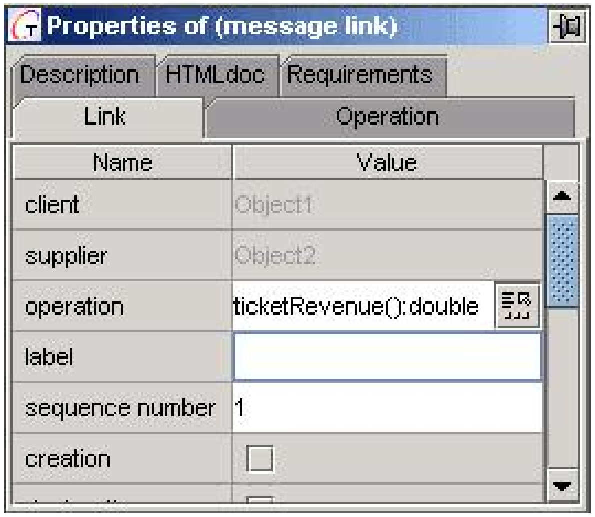
The message properties inspector is near the top of the message speedmenu.

When you specify a new operation in the inspector, Together creates a dialog box like the one below. For this problem, you should click the Create button.



Operation ticketRevenue():double not found in class Flight.
Create new operation ticketRevenue():double?

If you rename an existing method, Together's dialog box has three buttons: Rename, Create, and Cancel.



Name	Value
client	Object1
supplier	Object2
operation	ticketRevenue():double
label	
sequence number	1
creation	□

Putting control code on diagrams

In this section, we will change the activation bar on the **Reservation** lifeline to represent finding the revenue from a single reservation:

> "The revenue generated for a reservation is the price of the ticket if the reservation has been ticketed. If it has not been ticketed, the revenue is 0."

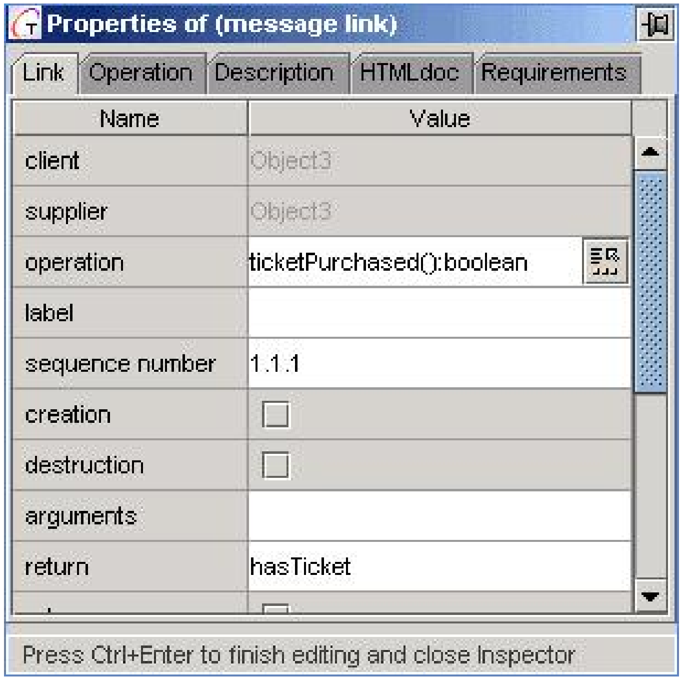
Step: Put a self message on the activation bar on the **Reservation** lifeline. Choose **ticketPurchased()** for its operation. Set the return value to **hasTicket**.

The sequence diagram toolbar has a self-message button (⮐). After you create the self-message, choose **ticketPurchased( )** as its operation.

For the remainder of this step, you'll need the properties inspector.

The return on the Link tab of the property inspector indicates the name of the value to be returned. Together can use that information to generate code. For our case, the code that Together will generate on demand is:

boolean hasTicket = this.ticketPurchased();



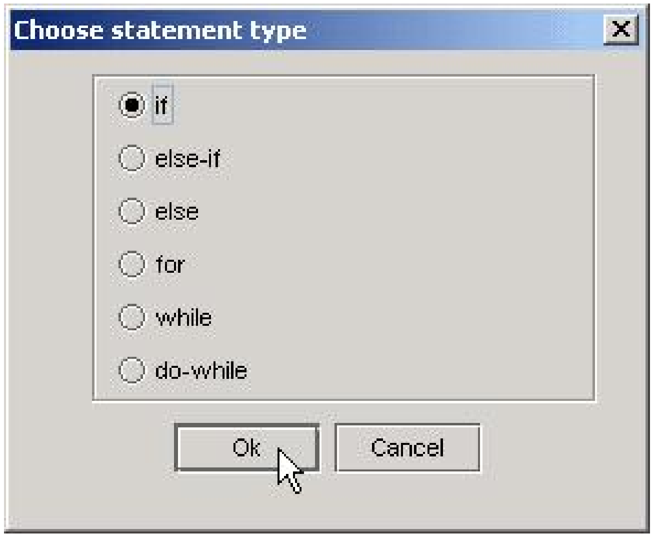
Name	Value
client	Object3
supplier	Object3
operation	ticketPurchased():boolean
label	
sequence number	1.1.1
creation	□
destruction	□
arguments	
return	hasTicket

Press Ctrl+Enter to finish editing and close Inspector

Step: Create two statement blocks on the activation bar on the **Reservation** lifeline.

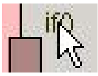
1. Put an **if** statement block on the activation bar beneath the self message.
2. Set the **if** condition to **hasTicket**.
3. Put an **else** statement block beneath the **if**.

Clicking the statement block button (**{ }**) and then clicking the **Reservation** activation bar brings up the dialog box here.

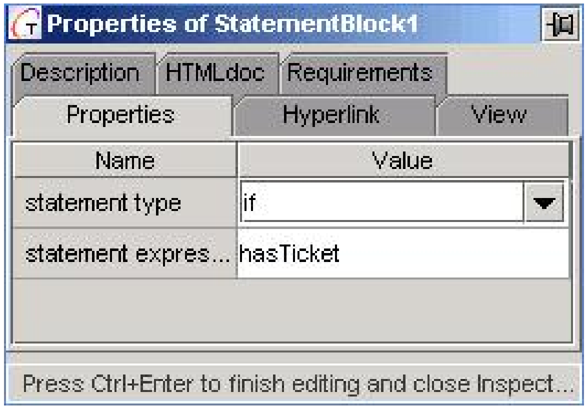


Choose statement type

● if
○ else-if
○ else
○ for
○ while
○ do-while



A statement block (such as an **if**) shows as a dark rectangle on the activation bar. Statement blocks have speedmenus that you can access by right-clicking statements or rectangles.



Properties of StatementBlock1

Name	Value
statement type	if
statement expres...	hasTicket

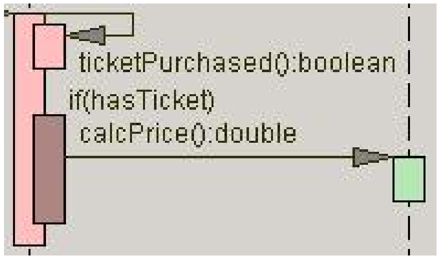
Press Ctrl+Enter to finish editing and close Inspect...

Set the **if** condition with its inspector. The condition goes in the "statement expression" textfield.

Step: Create a **calcPrice()** message from the **if** block to the **Ticket**.

Statement blocks, activation bars, and lifelines can be used for message starting and stopping points. For this step, be careful to start the message inside the **if** block rather than elsewhere on the activation bar.

In our diagram, we pulled the **if** block down the activation bar to allow more room for annotations.



Implementing code from diagrams

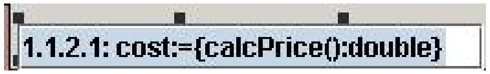
Together will generate code for a method using a sequence diagram if the method has no code in its body to start. *Before beginning the next step, remove any return statement (or other code) that you may have placed inside the body of **Reservation.calcPrice()**.*

Step: Implement **Reservation.calcPrice()**.

This is an easy step. After clearing out any code from the body of **Reservation.calcPrice()**, select **Generate Implementation** from the sequence diagram speedmenu. You should get two messages in the Message pane. The first for **Reservation.ticketPurchased()** and the second is for **Ticket.calcPrice()**.

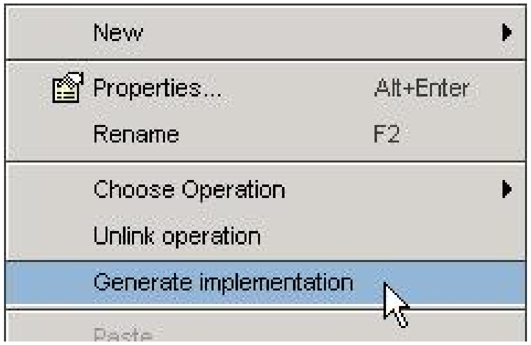
1. message #1.1.1 associated operation is not empty -- can't generate code
2. message #1.1.2.1 associated operation has no body-- can't generate code

Together will not generate any code for methods with non-empty bodies. Neither will it generate code for abstract methods.

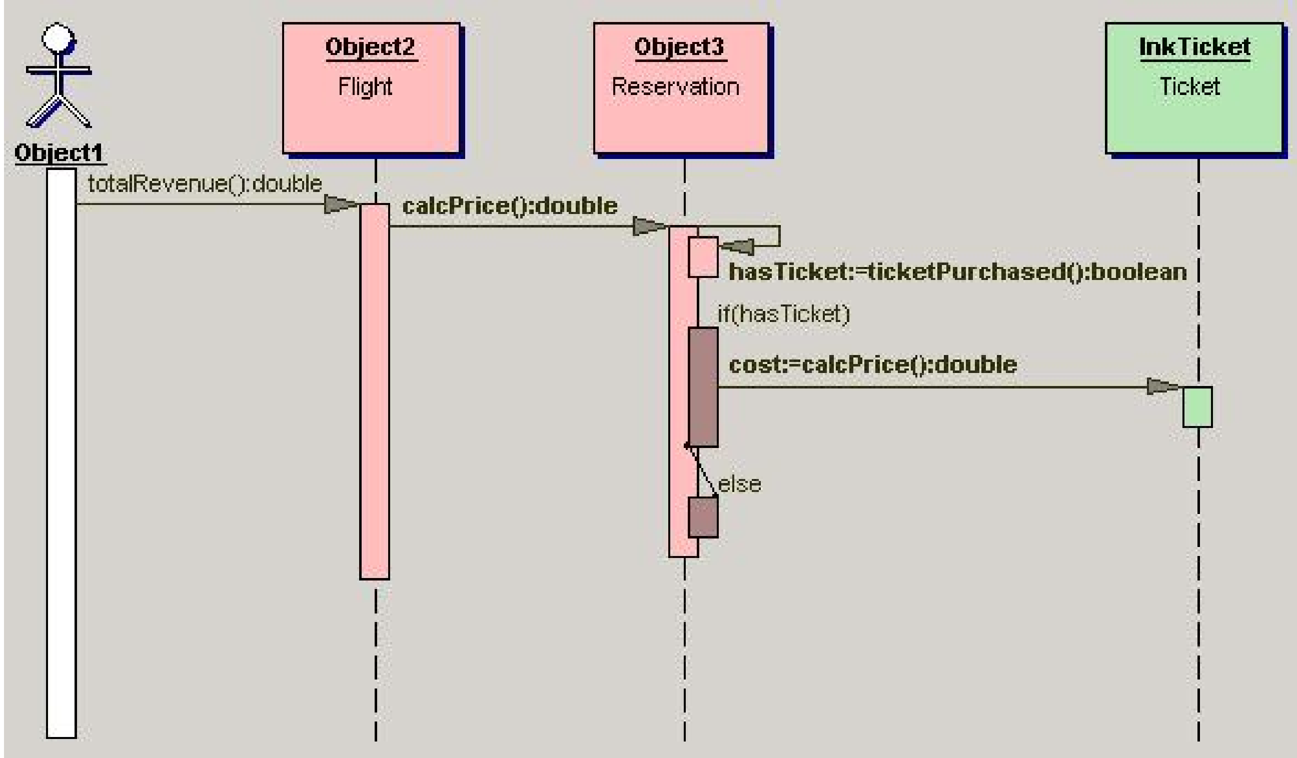


1.1.2.1: cost:={calcPrice():double}

#1.1.1 is a message sequence number. You can get sequence numbers from the message properties inspectors. You can also get them by double clicking the messages in the diagram.



Below is the completed sequence diagram, with three messages in boldface. Together sequence diagrams put boldface type on messages that are invoked as a result of creating code via **Generate Implementation**.



Sequence diagrams can give structure to code. But it's up to you to finish the details. Below is the incomplete code for **Reservation.calcPrice()**.

```
public double calcPrice(){
   // message #1.1.1 to this:AirlinePD.Reservation
   boolean hasTicket = this.ticketPurchased();
   if (hasTicket) {
     // message #1.1.2.1 to lnkTicket:AirlinePD.Ticket
         double cost = lnkTicket.calcPrice();
   }
   else { }
}
```

You can correct the code with two return statements:

1. Put a **return** statement inside the **if** statement block: **return cost;**
2. Put a **return** statement in the **else**:   **return 0;**

Step: Remove any syntax errors from **Reservation.calcPrice()** and **Flight.ticketRevenue()**. Correct logic errors as you see fit.

Creating sequence diagrams from existing code

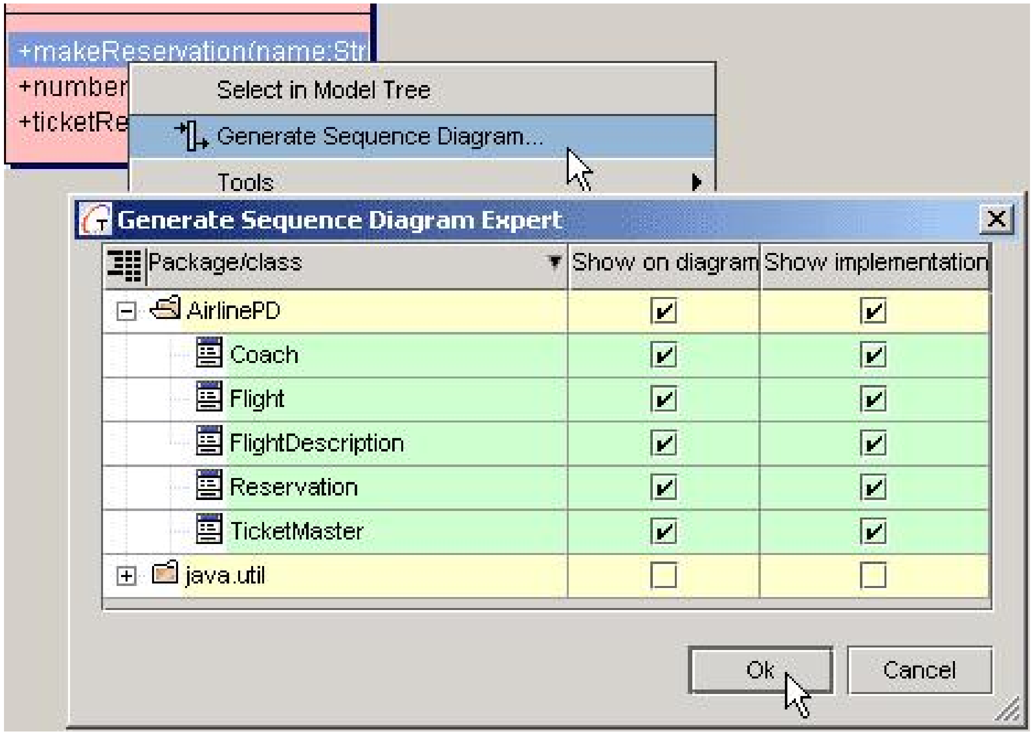
Together can generate sequence diagrams from code that has no syntax errors. As a preliminary step to your next task, you should Make the project as an error check.

Step: Generate a new sequence diagram from **Flight.makeReservation()**. Show every class from the **AirlinePD** package. But don't show anything from **java.util**.

The option to Generate Sequence Diagram is on operation speedmenus in class diagrams.

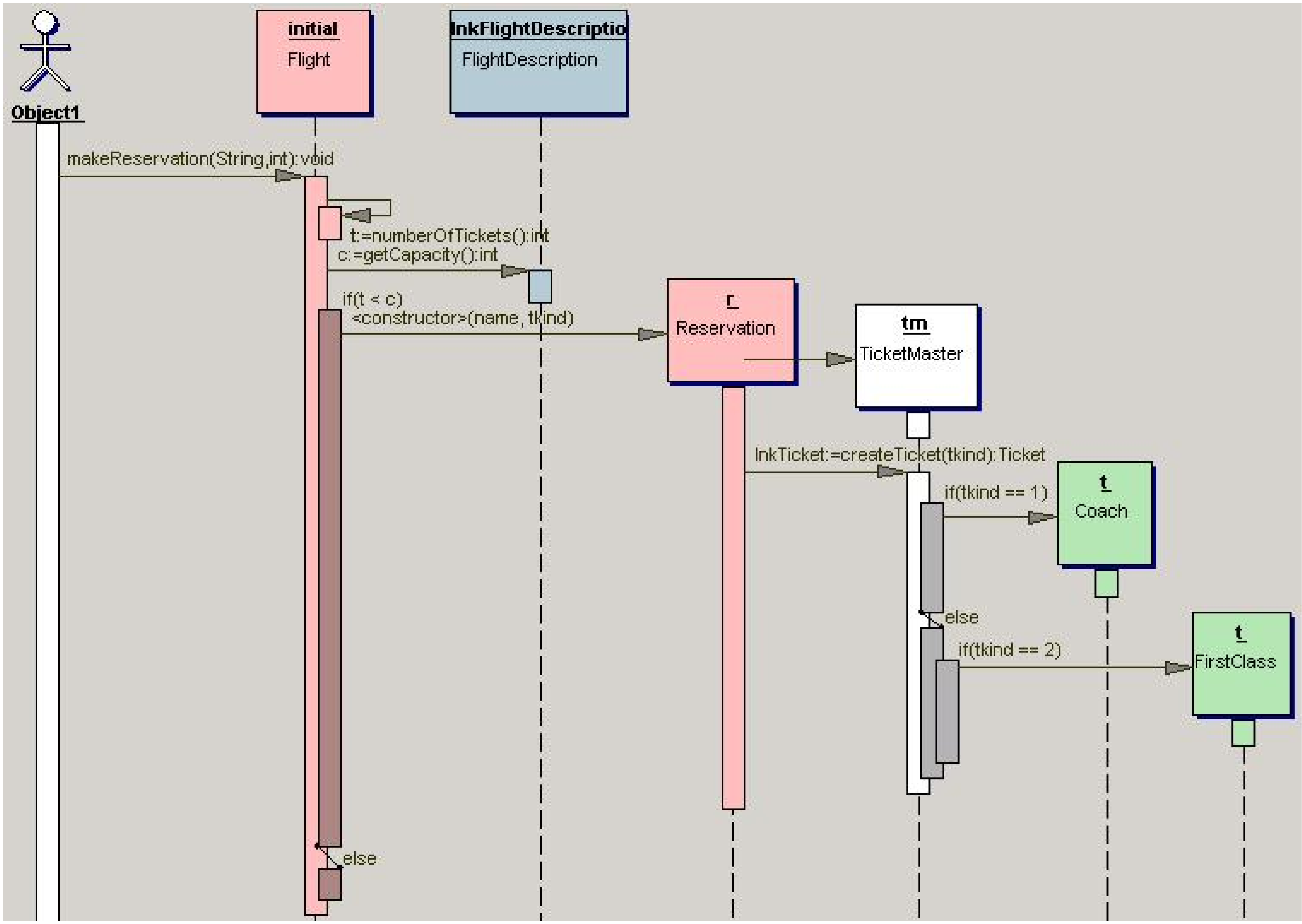
The snapshot to the right shows part of the speedmenu for **Flight.makeReservation()**.

The Generate Sequence Diagram Expert gives a choice of which classes and implementation details to show. For our sequence diagram, we checked off all the **java.util** items; we checked on all the **AirlinePD** items.


+makeReservation(name:Str
+number
+ticketRe
Select in Model Tree
Generate Sequence Diagram...
Tools
Generate Sequence Diagram Expert
Ok
Cancel

Package/class	Show on diagram	Show implementation
AirlinePD	☑	☑
Coach	☑	☑
Flight	☑	☑
FlightDescription	☑	☑
Reservation	☑	☑
TicketMaster	☑	☑
java.util	☐	☐

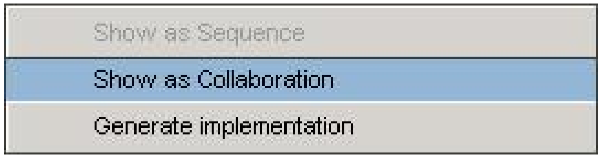
Below is our sequence diagram. Its (default) name is **Flight.makeReservation(1)**. Light rectangles are activation bars (corresponding to method calls). Dark rectangle correspond to loop or conditional statements. The final four objects are lowered on the diagram to indicate that they are created as the reservation is made.



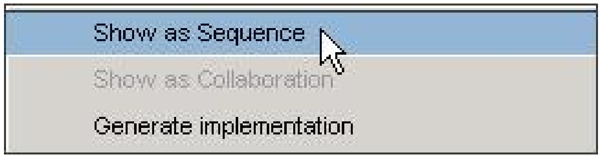
Step: Generate a collaboration diagram from **Flight.makeReservation(1)**.

Collaboration diagrams are equivalent to sequence diagrams. To switch from one to another, go to the diagram speedmenu. (Together maintains only one file for the two diagrams, representing different views of the same information.)

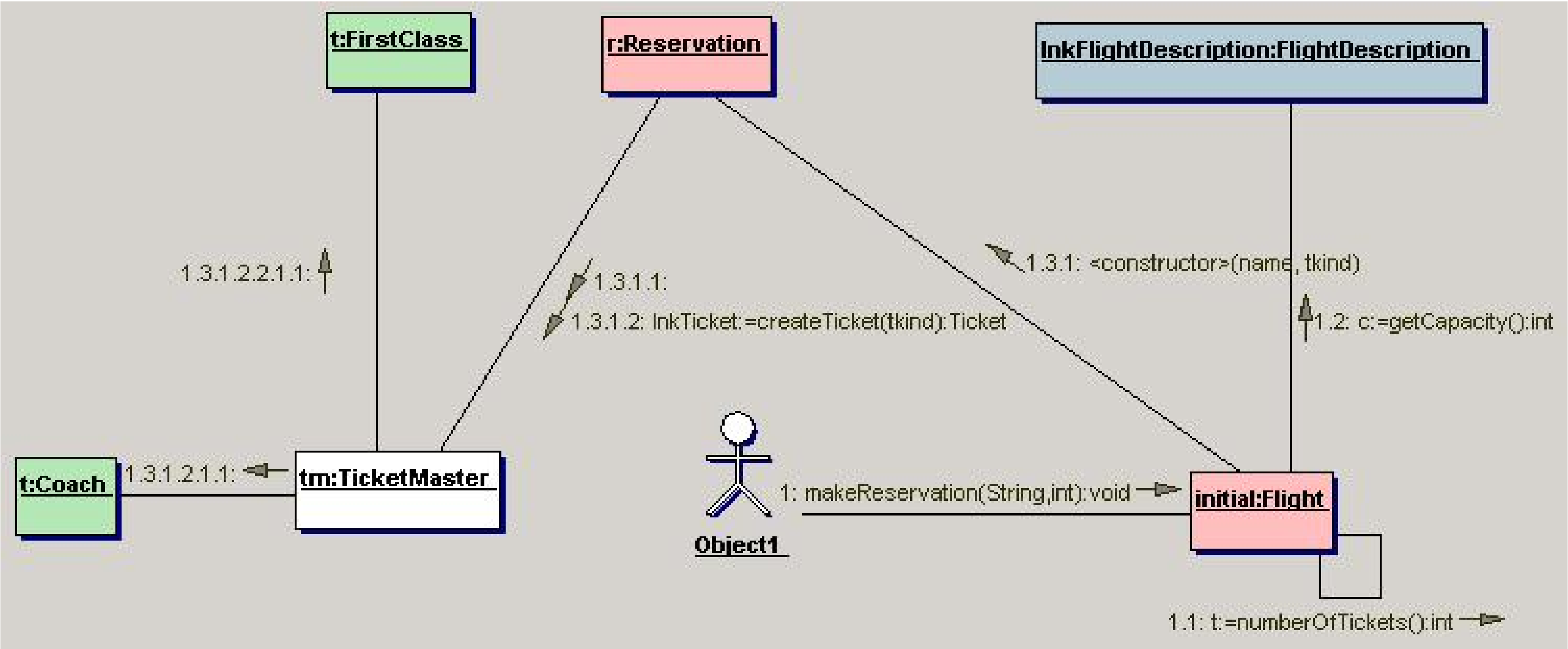
Speedmenu for sequence diagram



Speedmenu for collaboration diagram



Below is our collaboration diagram for **Flight.makeReservation(1)**. (We moved the nodes around to make them fit in a smaller space.)


t:FirstClass
r:Reservation
lnkFlightDescription:FlightDescription
1.3.1.2.2.1.1:
1.3.1.1:
1.3.1.2: lnkTicket:=createTicket(tkind):Ticket
1.3.1: <constructor>(name, tkind)
1.2: c:=getCapacity():int
t:Coach
1.3.1.2.1.1:
tm:TicketMaster
Object1
1: makeReservation(String,int):void
initial:Flight
1.1: t:=numberOfTickets():int

Hyperlinking project elements

Hyperlinks between Together objects (such as diagrams and diagram elements) can tie objects together and shortcut project navigation. When an object on a diagram is hyperlinked to another, its name appears in blue.

When you create as sequence diagram from an operation, Together automatically hyperlinks the operation to the sequence diagram. Look at your **AirlinePD** class diagram. The operation named **Flight.makeReservation** should be blue because it's hyperlinked to the collaboration diagram **Flight.makeReservation(1)**.

You can also create hyperlinks from one object to another directly via the object property inspector.

Step: Create a hyperlink from the **Make a reservation** use case (on the **MakeReservation** use case diagram) to these elements:

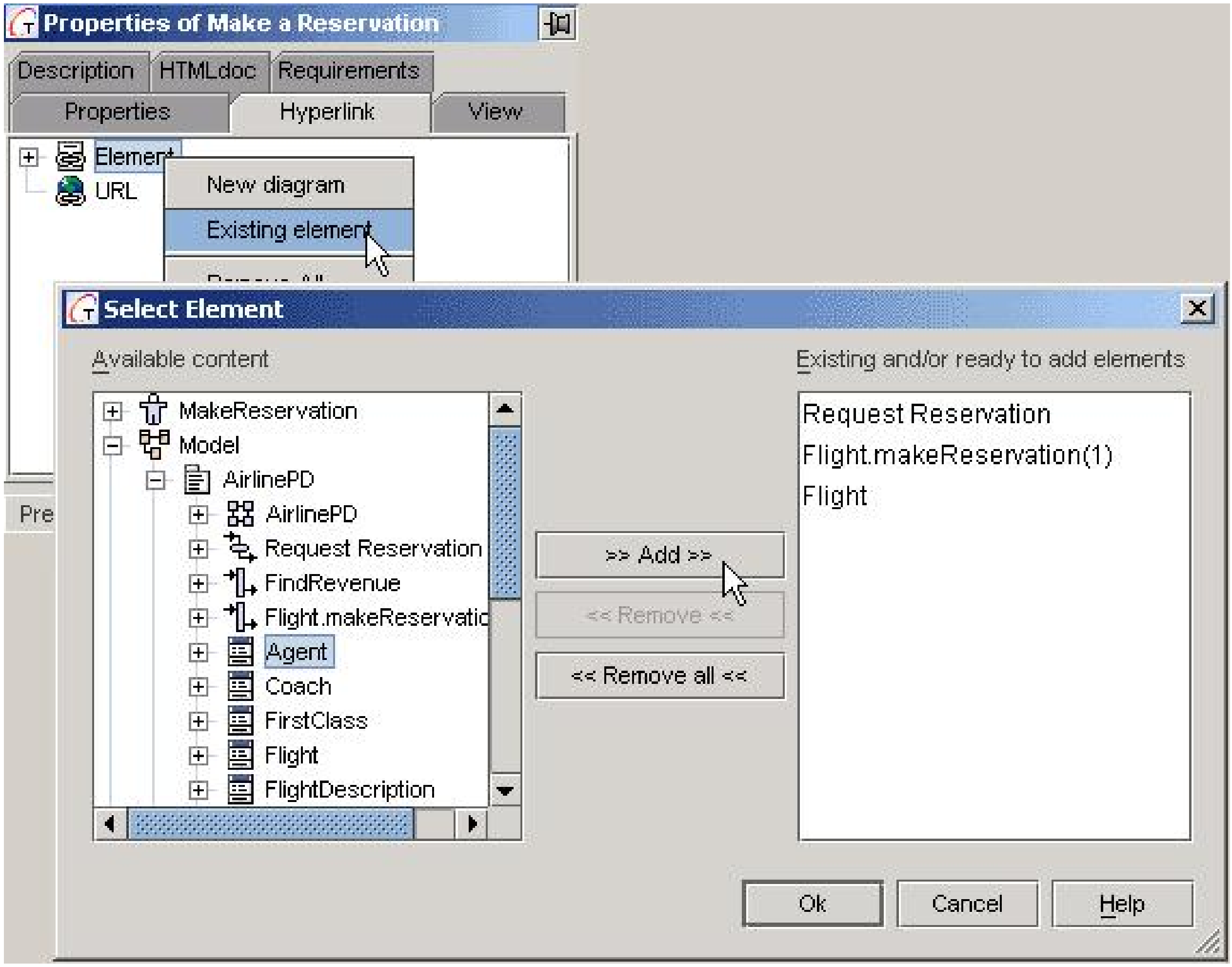
1. to the collaboration diagram **Flight.makeReservation(1)**
2. to the activity diagram, **Request Reservation**
3. to the **Flight** class on the **AirlinePD** class diagram
4. to the **Agent** class on the **AirlinePD** class diagram

Start at the **MakeReservation** use case diagram.

Right click the **Make a Reservation** use case to bring up its speedmenu and properties inspector. Then go to the Hyperlink tab in the inspector.

Right clicking on Element brings a Select Element menu giving a choice of hyperlink items to add as hyperlinks.

When you finish, the **Make a Reservation** use case should display in blue font.

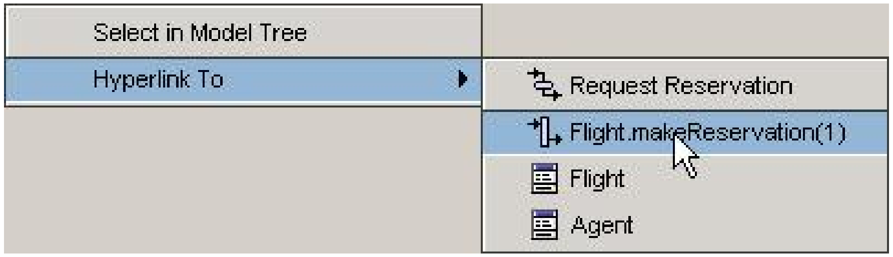


Request Reservation
Flight.makeReservation(1)
Flight

Step: Travel from the **Make a Reservation** use case to the **makeReservation(1)** diagram via the new hyperlink.

You can travel on the **Make a Reservation** use case hyperlinks via its speedmenu. (The cursor does not change into a familiar hand when it is above a hyperlinked element because there are several possible destinations.)

Use the forward (⇨) and reverse (⇦) arrows on the Main toolbar to travel back and forth on hyperlinks.



Tips and Tricks

- You can hyperlink from a Together element to items outside Together entirely. For example, you may want to hyperlink a use case to a requirements document.
- Sequence diagrams are tied intimately to code, but Together keeps code and sequence diagrams in sync only on demand.
- You can use the in-place editor for messages to change the name of the operation but not its return type.
- To create a new non-void operation for a message, use the message inspector rather than the top item on the speedmenu.

← PREVIOUS | TOP ↑ | NEXT → | START HOME

Last Revised: Thu, Apr 12, 2001

Together Tutorial
Part 9: Documentation Generation

Self-documenting code may be an oxymoron. But with Together, self-documenting projects become a reality. Together will generate documentation -- all kinds of documentation, from HTML to RTF to PDF. The documentation is fully hyperlinked to show relationships among project entities. For anyone needing customized documentation in a special format, Together even offers a powerful document designer.

In this part of the Together Tutorial we'll show you how to generate documentation. The discussion is short, in part because generating documentation is so easy.

Contents:

Generating HTML documentation

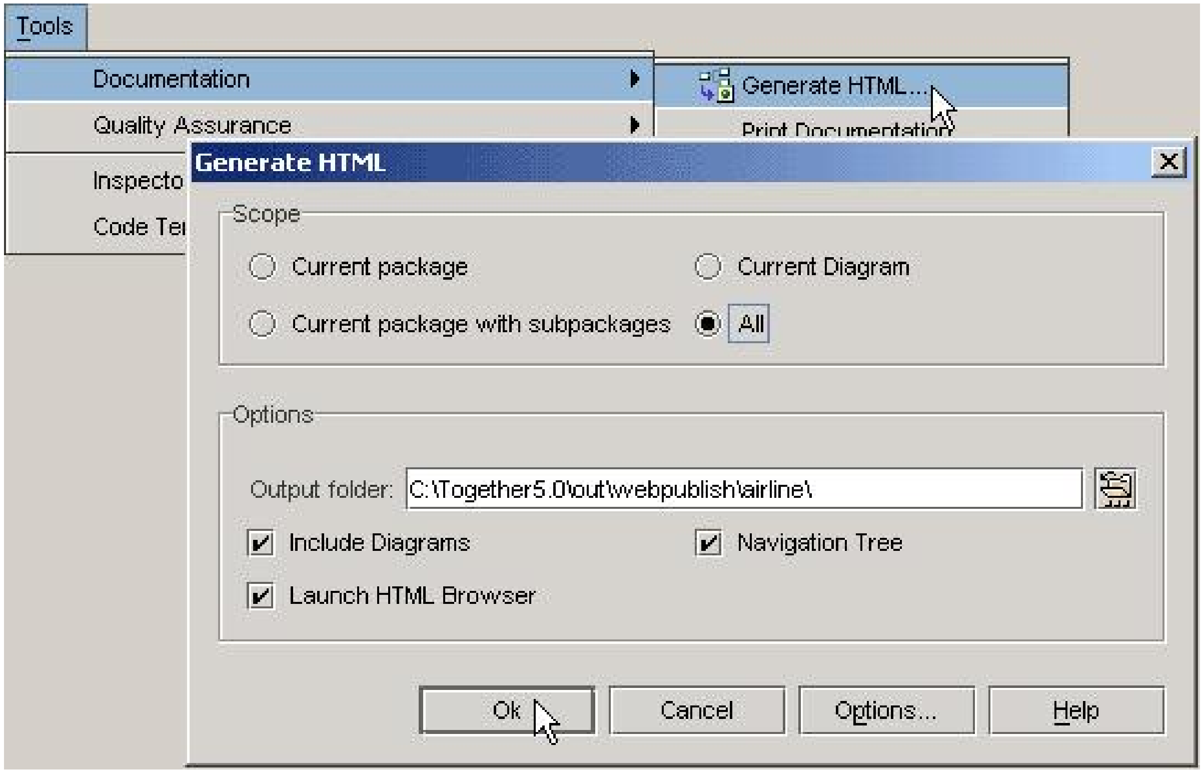
Together uses Javadoc comments in code to keep track of properties of diagrams and diagram elements. It uses those Javadoc comments in creating documentation as well.

Step: Generate HTML documentation for the entire **airline** project.

HTML documentation generation is a simple click of a button. Select **Tools|Documentation|Generate HTML** from the Main menu.

With the options button on the bottom of the dialog box, you can specify a variety of settings:

1. Whether to include or exclude header tag information (author, version, etc.)
2. Special HTML options (window title, stylesheets, etc.)
3. The visibility levels of classes shown in the documentation



Scope

○ Current package ○ Current Diagram

○ Current package with subpackages ● All

Options

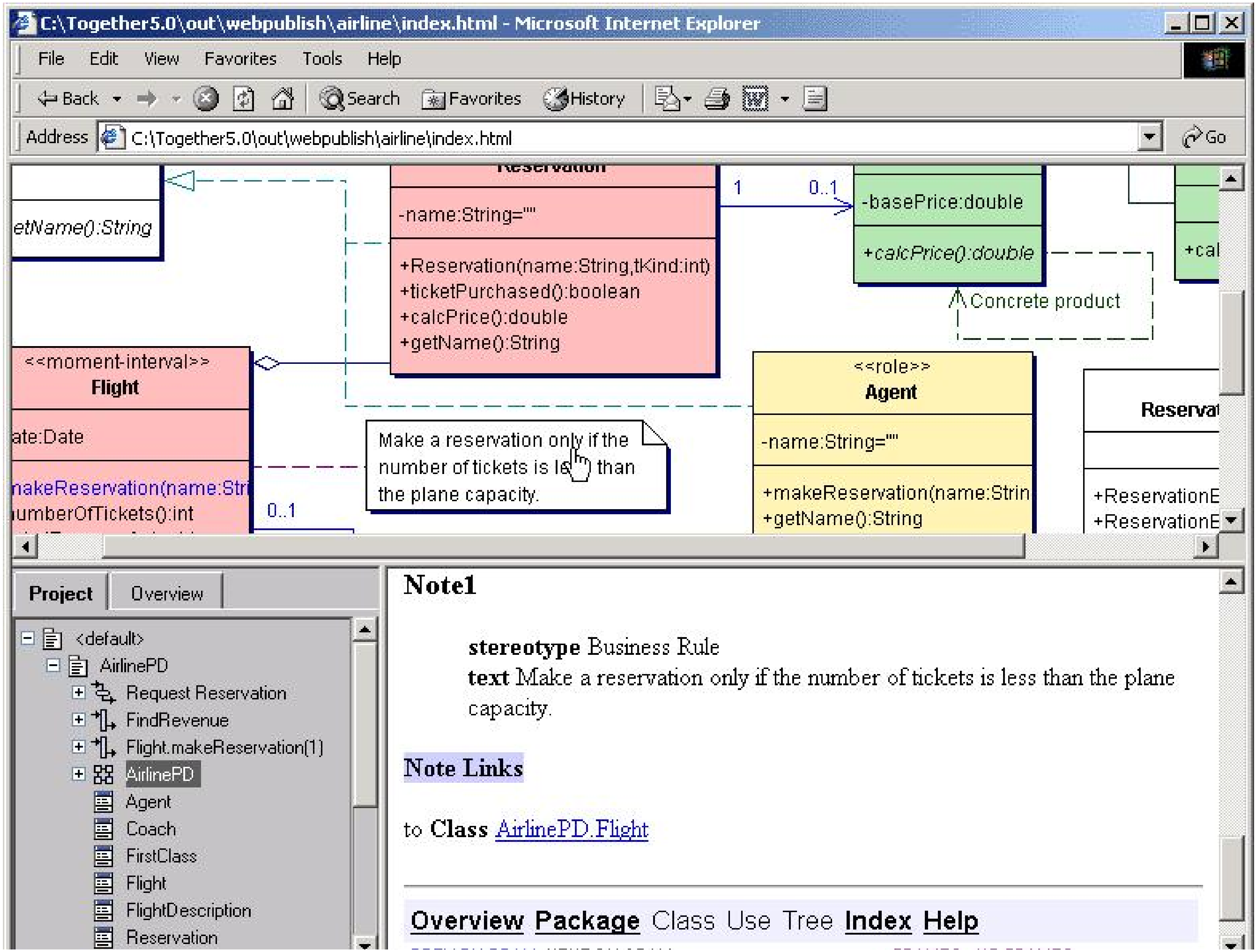
Output folder: C:\Together5.0\out\webpublish\airline\

☑ Include Diagrams ☑ Navigation Tree

☑ Launch HTML Browser

The entire HTML document is hyperlinked, with both terms and image maps. When you click on an item in a class diagram, for example, its documentation shows in the lower right frame.

Below is a snapshot of our documentation. The browser displays the resulting documentation in three frames. The top frame is a diagram. The lower left is an applet with an explorer and overview tabs. The lower right frame has the written documentation.


C:\Together5.0\out\webpublish\airline\index.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\Together5.0\out\webpublish\airline\index.html
Go
etName():String
Reservation
-name:String=""
+Reservation(name:String,tKind:int)
+ticketPurchased():boolean
+calcPrice():double
+getName():String
1
0..1
-basePrice:double
+calcPrice():double
+cal
Concrete product
<<moment-interval>>
Flight
ate:Date
nakeReservation(name:Stri
umberOfTickets():int
0..1
Make a reservation only if the number of tickets is le than the plane capacity.
<<role>>
Agent
-name:String=""
+makeReservation(name:Strin
+getName():String
Reservat
+ReservationE
+ReservationE
Project
Overview
<default>
AirlinePD
Request Reservation
FindRevenue
Flight.makeReservation(1)
AirlinePD
Agent
Coach
FirstClass
Flight
FlightDescription
Reservation

Note1

stereotype Business Rule
text Make a reservation only if the number of tickets is less than the plane capacity.

Note Links

to **Class** AirlinePD.Flight

Overview **Package** Class Use Tree **Index** **Help**

file:///C:/Together5.0/out/webpublish/airline/AirlinePD/AirlinePD.cl.html#a0

Notice that the business rule requirement on the note shows up as part of the documentation.

Generating documentation in RTF format

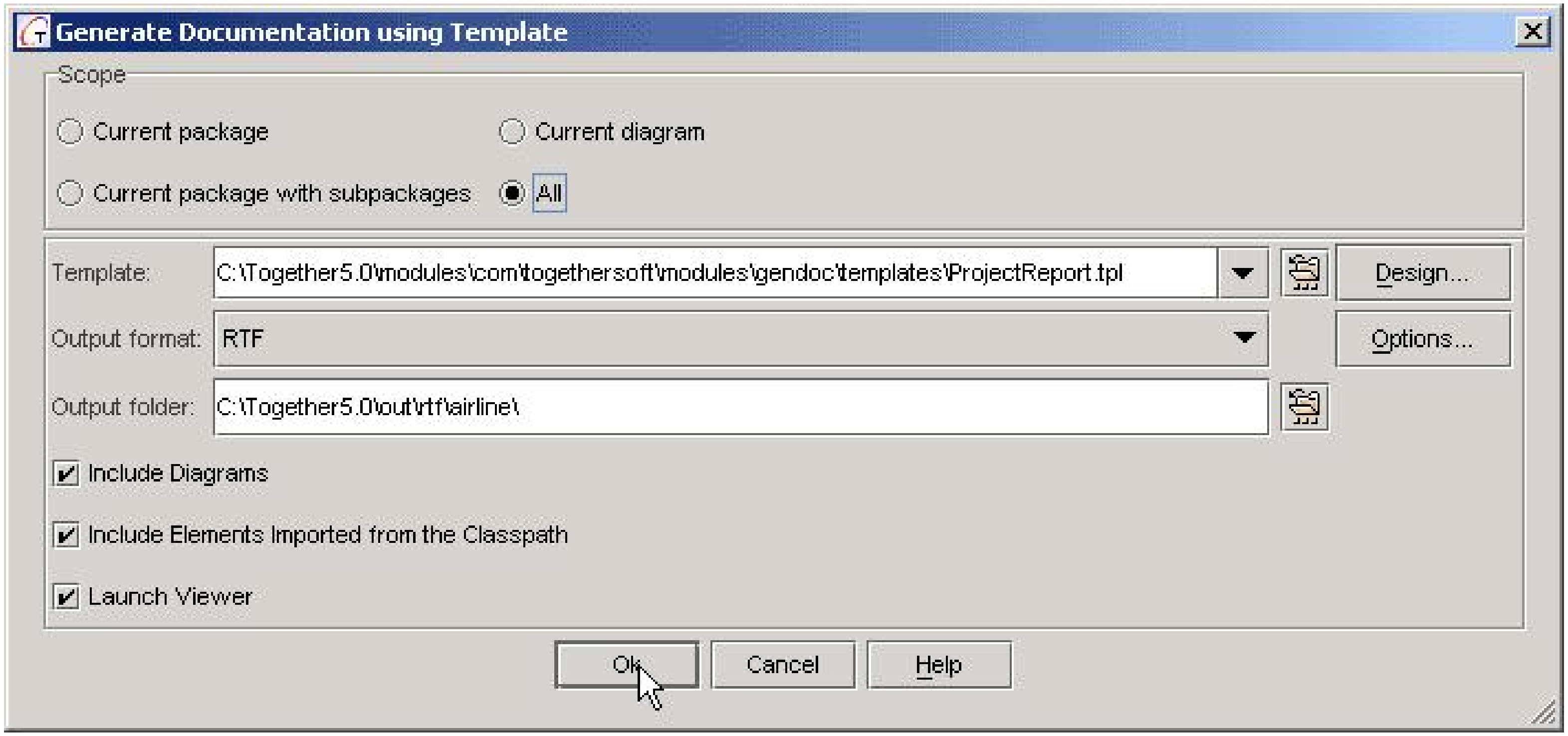
Together generates documentation in several formats: text, HTML, PDF, and RTF.

Step: Generate RTF documentation for the entire **airline** project.

Together has a template for documentation in RTF, accessible via the Main menu:

Tools|Documentation|Generate using Template.

The resulting dialog box has a familiar look.



Generate Documentation using Template

Scope

○ Current package ○ Current diagram

○ Current package with subpackages ● All

Template: C:\Together5.0\modules\com\togethersoft\modules\gendoc\templates\ProjectReport.tpl

Output format: RTF

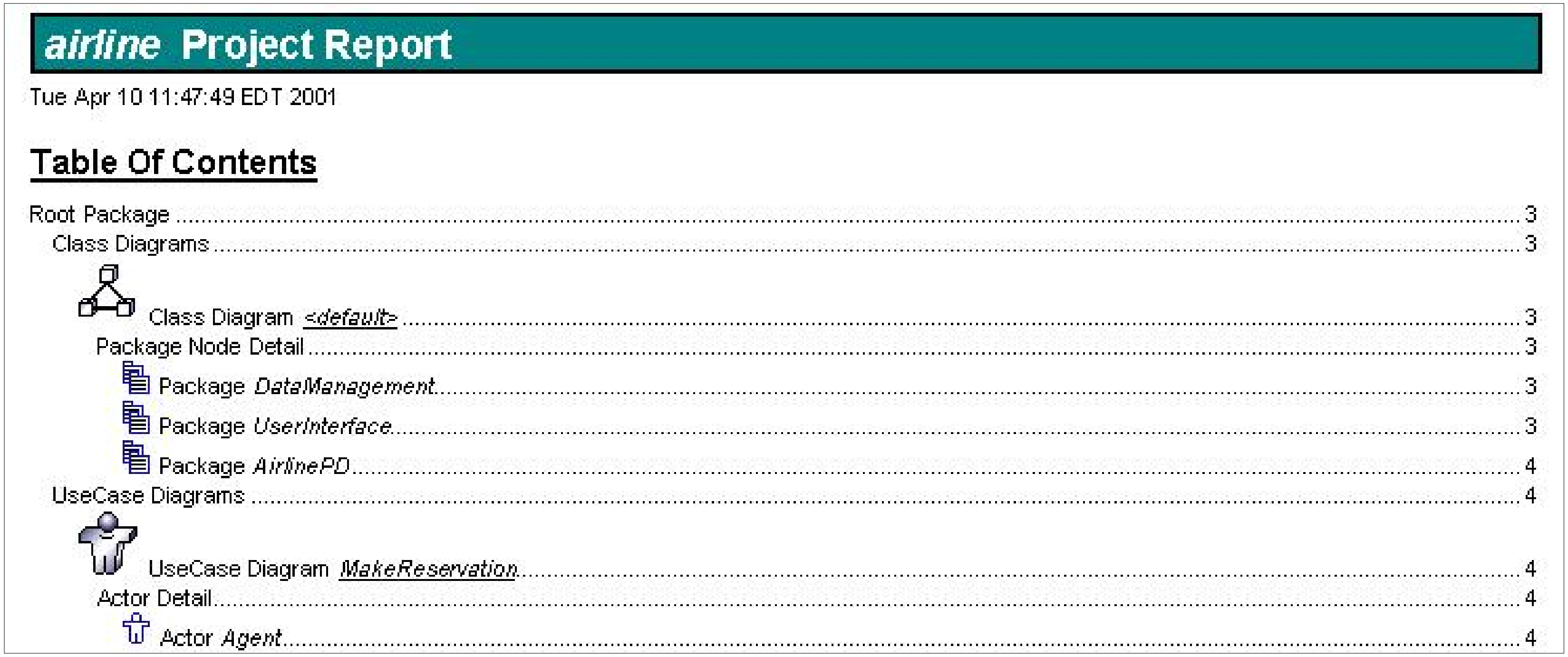
Output folder: C:\Together5.0\out\rtf\airline\

☑ Include Diagrams

☑ Include Elements Imported from the Classpath

☑ Launch Viewer

Below is the first part of the first of 26 pages of documentation that Together generated.



airline Project Report

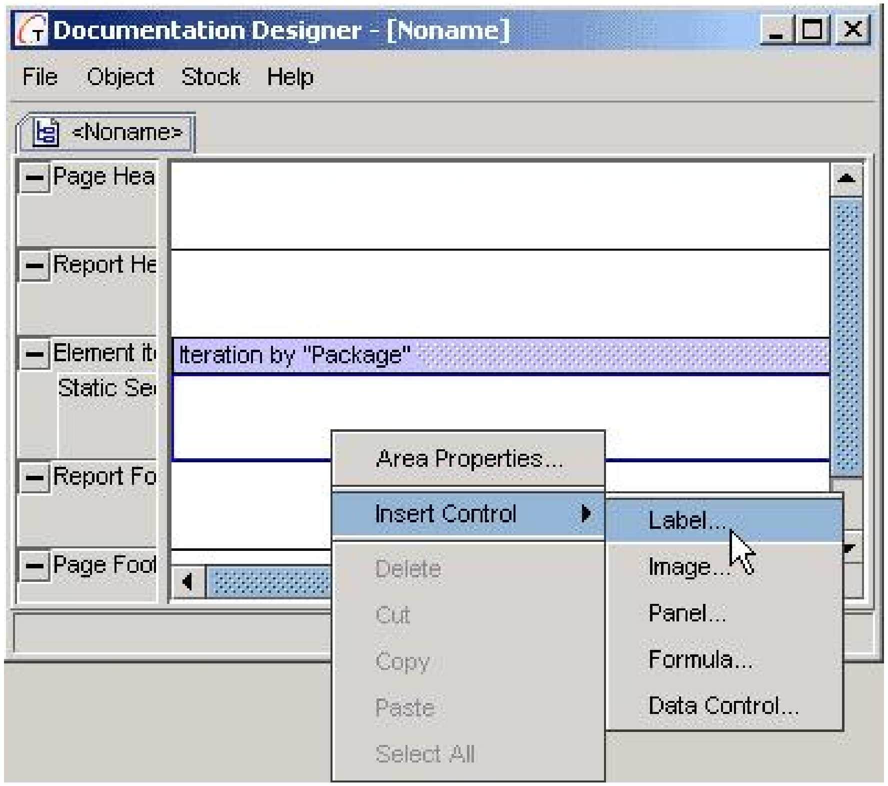
Tue Apr 10 11:47:49 EDT 2001

Table Of Contents

Together ControlCenter lets you create your own documentation template. Look at the Main menu under **Tools|Documentation|Design Template**.

On the right is snapshot of a new template in the documentation designer.

With the document designer, you can specify headers and footers as well as the internal organization of the document.



To get our screen snapshot above, we started with **File|New Document Template** in the **Documentation Designer**. Then we went through two steps:

1. We changed the first **Element Iterator** to Package via right-clicking its purple bar. Together lets you iterate over a large variety of types, from Activity and Activity Diagrams to XML Structure Diagrams.
2. We right-clicked the **Static Section** immediately below. The items on the **Insert Control** menu are described in the Together documentation:
   - Labels -- text labels
   - Images -- gifs, etc.
   - Panels -- for controlling presentation format
   - Formulas -- for data not available directly from the source
   - Data Control -- information that the report will actually display

Step: If you have Together ControlCenter, generate PDF documentation for the entire **airline** project.

Selecting **Tools|Documentation|Print** brings a dialog box with an option to create a PDF file. The dialog box has a convenient preview button.

Tips and Tricks

- Together uses many of the items on properties inspectors in generating documentation.
- When you have a choice of files or folders, Together will show the "pick folder" icon (📂). Click it to access the file system.
- Hyperlinking is not restricted to HTML documentation. It is also part of RTF documentation.
- Printed diagrams alone are a source of documentation. You can see where page breaks would come by checking **Show print grid** on the View tab of Diagram options. When you select **Layout|All for Printing** from a diagram speedmenu, Together will arrange not to split diagram items over page boundaries when possible.

← PREVIOUS | TOP ↑ | NEXT → | START HOME

Last Revised: Thu, Apr 12, 2001

Together Tutorial
Part 10: Audits and Metrics

Together provides built-in quality assurance features to help enforce company standards and conventions, capture real metrics, and improve what you do. Two features are specifically designed for quality assurance: audits and metrics. Audits check code for conformance to user-defined styles, maintenance and robustness guidelines. Metrics calculate the complexity of the code.

Together has long supported audits for Java projects. And now, Together 5.0 supports C++ audits as well.

The airline project is the basis for discussion in this part of the Together Tutorial. For a richer example, turn to the Cash Sales project that ships with Together.

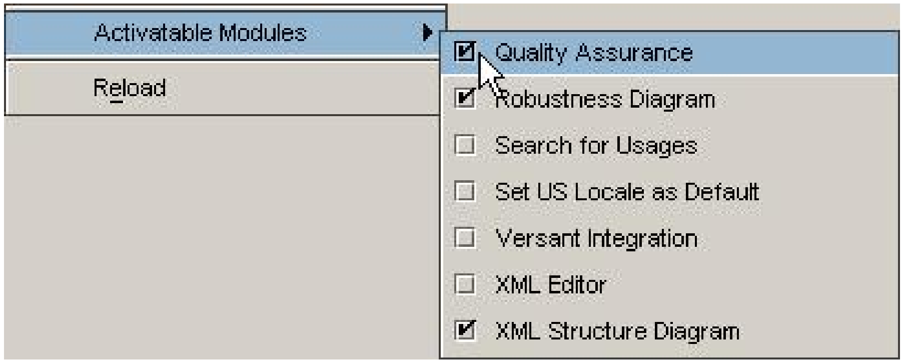
Contents:

Note: This is an optional part of the tutorial. Only Together ControlCenter supports audits and metrics.

Auditing your project

Together will generate Audits and metrics only if the Quality Assurance module is active. You can activate the module by checking on Quality Assurance from the **Options|Activatable Modules** item on the Main menu.

Start the next step from the default package.

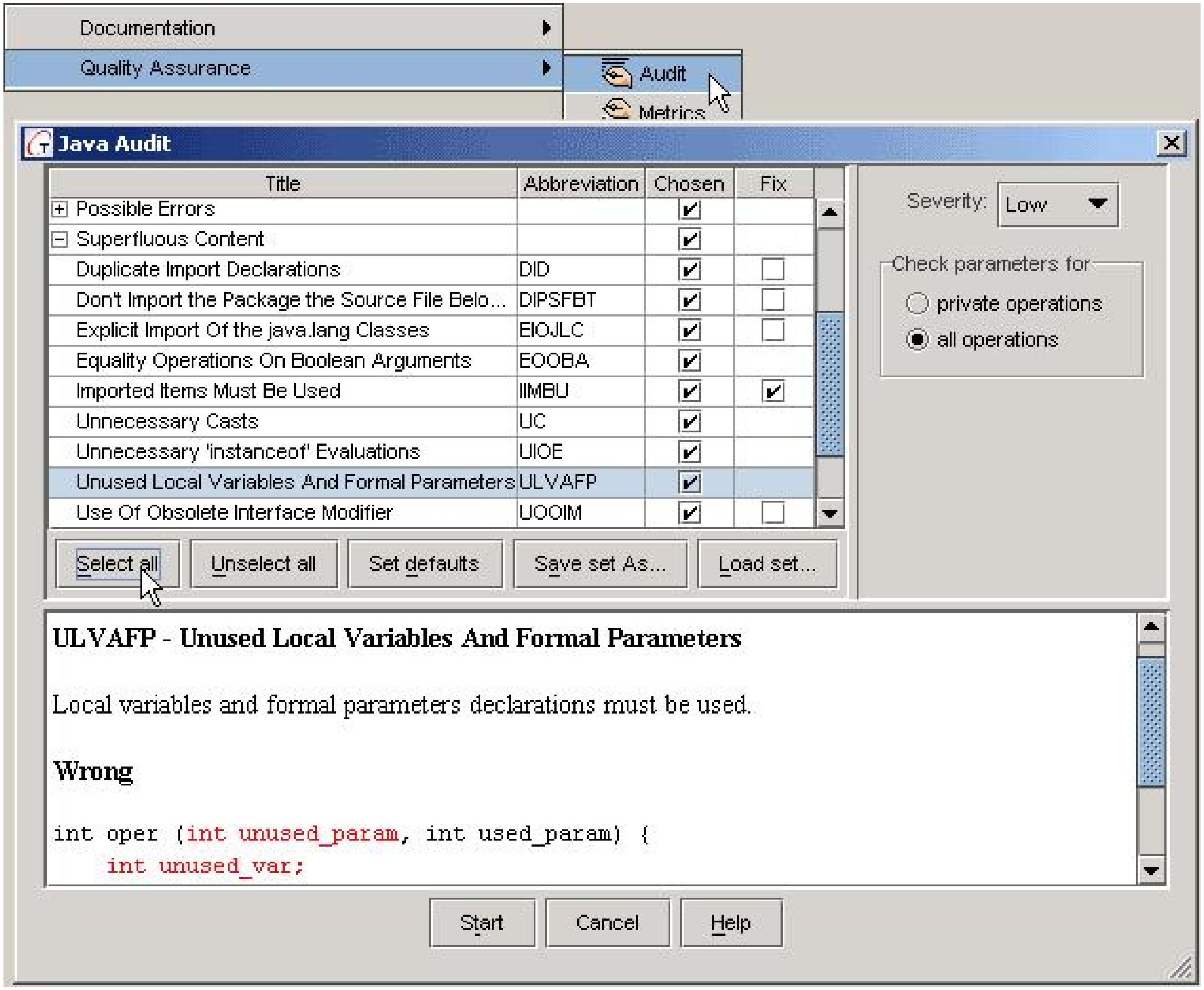


Step: Generate complete audits for the airline project.

Quality Assurance is listed on package diagram speedmenus and the **Tools** menu.

Auditing starts with a dialog box for selecting audit standards.

For this audit, click the **Select all** button. (You can also load sets of audits, including the Sun code conventions for Java, which are available from the **Load set** button.)



Title	Abbreviation	Chosen	Fix
⊞ Possible Errors		☑	
⊟ Superfluous Content		☑	
Duplicate Import Declarations	DID	☑	□
Don't Import the Package the Source File Belo...	DIPSFBT	☑	□
Explicit Import Of the java.lang Classes	EIOJLC	☑	□
Equality Operations On Boolean Arguments	EOOBA	☑	
Imported Items Must Be Used	IIMBU	☑	☑
Unnecessary Casts	UC	☑	
Unnecessary 'instanceof' Evaluations	UIOE	☑	
Unused Local Variables And Formal Parameters	ULVAFP	☑	
Use Of Obsolete Interface Modifier	UOOIM	☑	□

Severity: Low

Check parameters for

○ private operations

● all operations

ULVAFP - Unused Local Variables And Formal Parameters

Local variables and formal parameters declarations must be used.

Wrong

```
int oper (int unused_param, int used_param) {
    int unused_var;
```

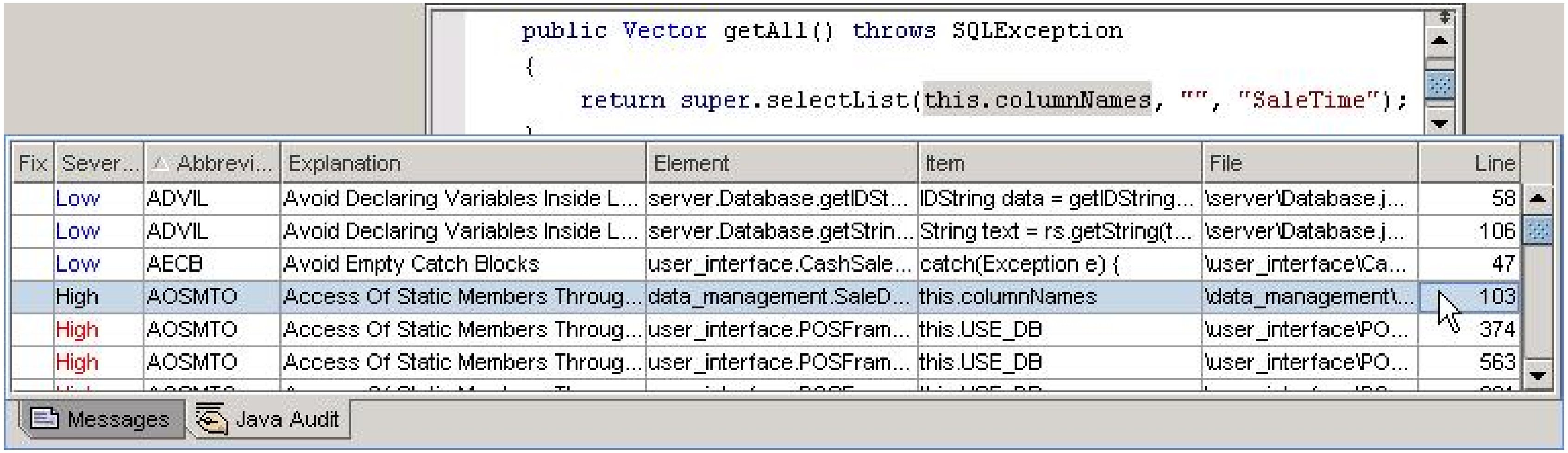
The scrolling list of the Java Audit window has a Chosen and Fix columns. The Chosen column lets you select or deselect specific audits. Checking a box on the Fix column instructs Together to change the source code to eliminate the problem. The upper right part of the Java Audit window lets you rank the standards as Low, Medium, or High priority. The resulting audit table can be sorted according to that rank.

The lower portion of the Java Audit window documents each audit standard. For most items, the documentation shows examples of code that violate the standard as well as equivalent code that adheres to the standard.

Generating audits on the airline project should be quick. Together brings up the results in a table in the Message pane.

- Click on any column header to sort the table according to the entries in that column.

- Click on any entry to bring up its speedmenu. The description of each entry is a choice on the speedmenu.
- Double click any line in the table to bring up the corresponding code in the Editor menu.



```
public Vector getAll() throws SQLException
{
    return super.selectList(this.columnNames, "", "SaleTime");
}
```

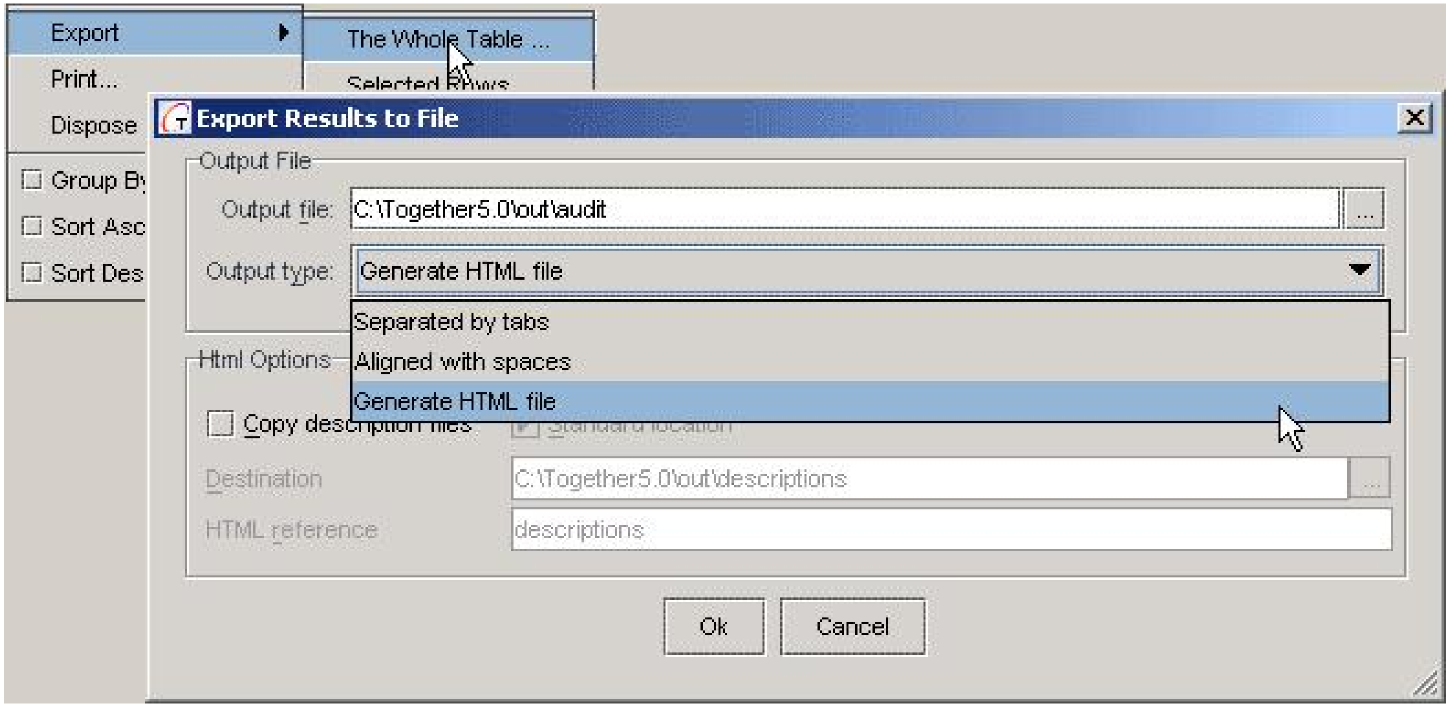
Fix	Sever...	Abbrevi...	Explanation	Element	Item	File	Line
	Low	ADVIL	Avoid Declaring Variables Inside L...	server.Database.getIDSt...	IDString data = getIDString...	\server\Database.j...	58
	Low	ADVIL	Avoid Declaring Variables Inside L...	server.Database.getStrin...	String text = rs.getString(t...	\server\Database.j...	106
	Low	AECB	Avoid Empty Catch Blocks	user_interface.CashSale...	catch(Exception e) {	\user_interface\Ca...	47
	High	AOSMTO	Access Of Static Members Throug...	data_management.SaleD...	this.columnNames	\data_management\...	103
	High	AOSMTO	Access Of Static Members Throug...	user_interface.POSFram...	this.USE_DB	\user_interface\PO...	374
	High	AOSMTO	Access Of Static Members Throug...	user_interface.POSFram...	this.USE_DB	\user_interface\PO...	563

Step: Create an HTML document from the audit table.

The table speedmenu has an **Export** command. For this step, choose **Export|The Whole Table**.

Together brings up a dialog box for selecting the file name and file type.

One of the HTML options copies links to the descriptions of each audit standard to the table. (Ours is unchecked.)



Output File

Output file: C:\Together5.0\out\audit

Output type: Generate HTML file

- Separated by tabs
- Aligned with spaces
- Generate HTML file

Html Options

☐ Copy description files ☑ Standard location

Destination: C:\Together5.0\out\descriptions

HTML reference: descriptions

Generating project metrics

You'll want to begin at the default diagram for the next step also.

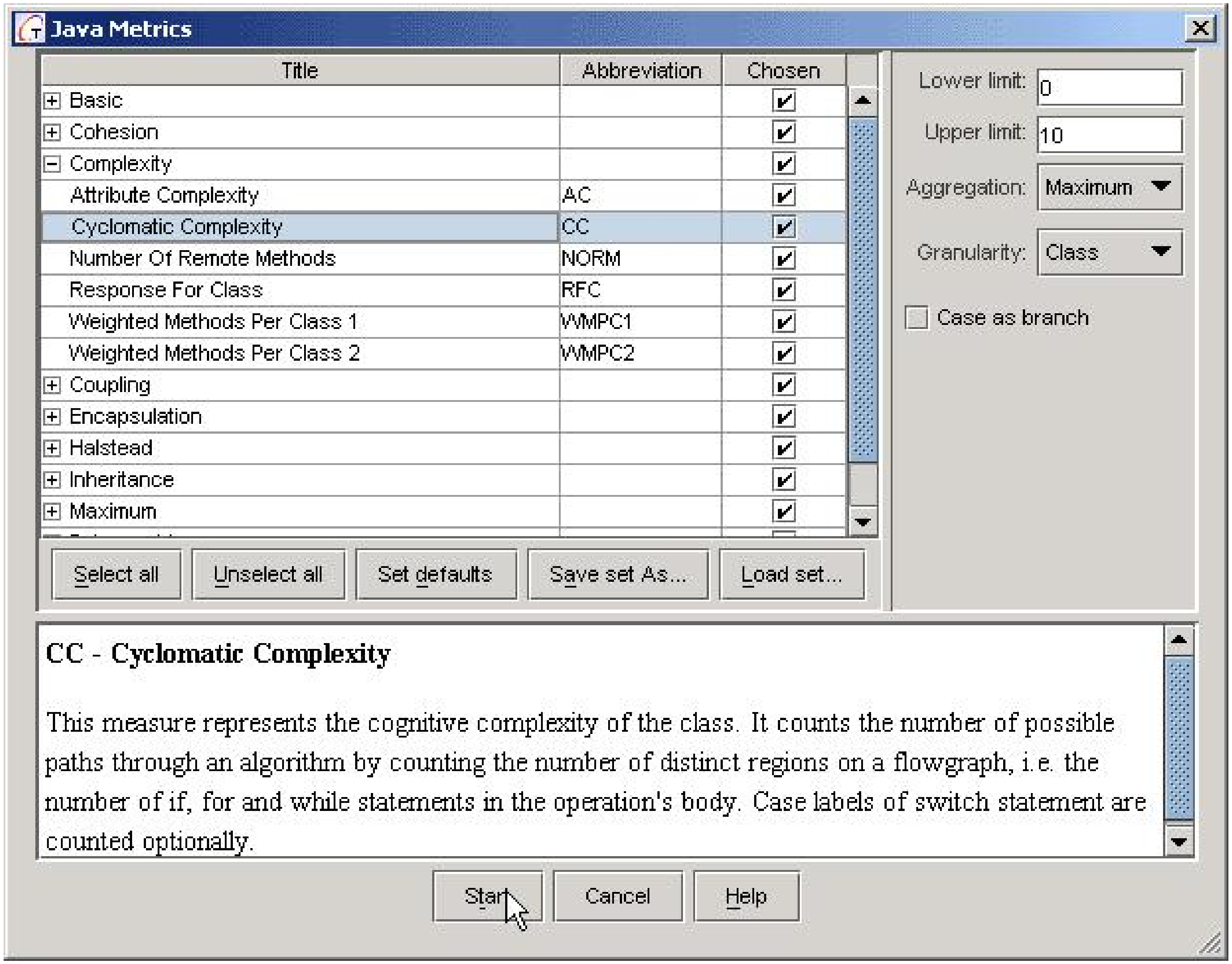
Step: Generate all metrics for the airline project.

Metrics are available from the same menu as Audits. From the diagram speedmenu, select **Quality Assurance|Metrics**.

The dialog window should look familiar. In the upper left is a list of the possible metrics. For this step, you should click the **Select all** button.

The panel in the upper right lets you select the upper and lower limits for each metric. The granularity can be according to class or operation.

The lower panel documents each metric.



Java Metrics

Title	Abbreviation	Chosen
⊞ Basic		☑
⊞ Cohesion		☑
⊟ Complexity		☑
Attribute Complexity	AC	☑
Cyclomatic Complexity	CC	☑
Number Of Remote Methods	NORM	☑
Response For Class	RFC	☑
Weighted Methods Per Class 1	WMPC1	☑
Weighted Methods Per Class 2	WMPC2	☑
⊞ Coupling		☑
⊞ Encapsulation		☑
⊞ Halstead		☑
⊞ Inheritance		☑
⊞ Maximum		☑

Lower limit: 0

Upper limit: 10

Aggregation: Maximum

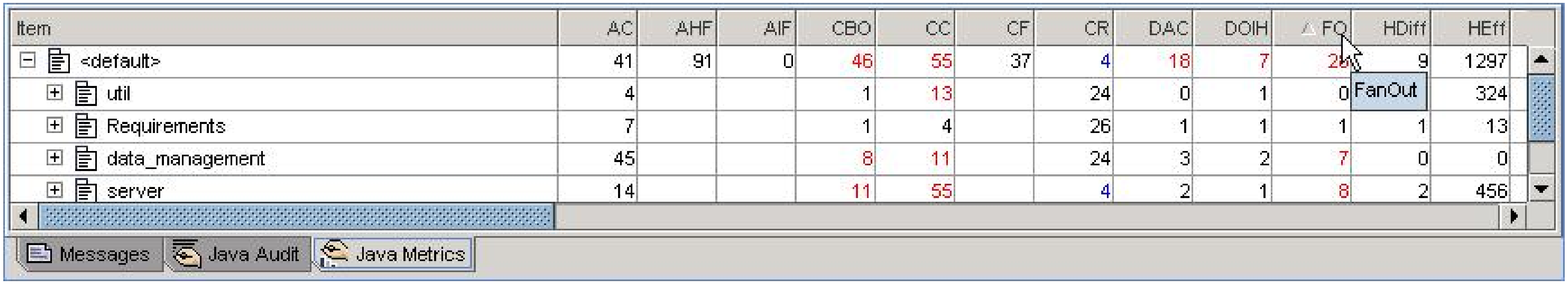
Granularity: Class

☐ Case as branch

CC - Cyclomatic Complexity

This measure represents the cognitive complexity of the class. It counts the number of possible paths through an algorithm by counting the number of distinct regions on a flowgraph, i.e. the number of if, for and while statements in the operation's body. Case labels of switch statement are counted optionally.

The resulting table in the Message pane is similar to the Audit table, with packages and classes marking the rows and metrics marking the columns. Each column heading has a flyover box with it unabbreviated name.

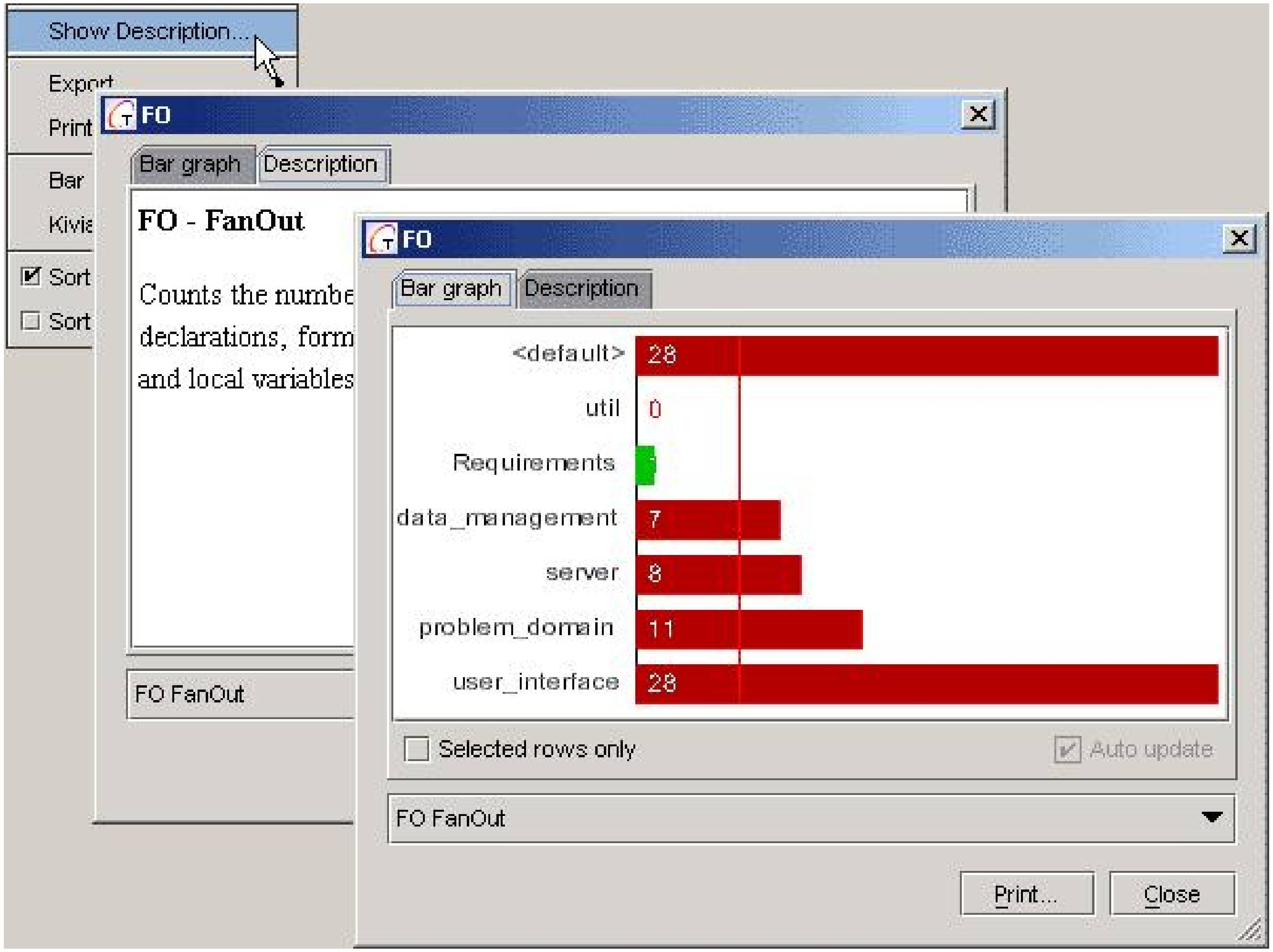


Item	AC	AHF	AIF	CBO	CC	CF	CR	DAC	DOIH	FO	HDiff	HEff
⊟ <default>	41	91	0	46	55	37	4	18	7	28	9	1297
⊞ util	4			1	13		24	0	1	0		324
⊞ Requirements	7			1	4		26	1	1	1	1	13
⊞ data_management	45			8	11		24	3	2	7	0	0
⊞ server	14			11	55		4	2	1	8	2	456

Step: Find the description of Fan Out. Generate a bar graph for the Fan Out metric on the Cash Sales project.

The speedmenu for each column gives access to a description of the column header.

The description window has two tabs, one for the description itself and the other for a bar graph.



FO - FanOut

Counts the numbe declarations, form and local variables

FO FanOut

☐ Selected rows only ☑ Auto update

FO FanOut

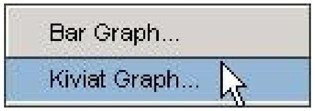
Step: Reset these metric limits for the Cash Sales project.

- CBO -- Coupling Between Objects -- (Upper Limit = 5)
- CR -- Comment Ratio -- (Lower Limit = 15)
- FO -- Fan Out -- (Upper Limit = 3).

Generate the metrics. Then create a Kiviat graph for **data_management**.

You'll have to start from the beginning to reset the metric limits. The upper right panel of the Metrics dialog box has fields for entering new limits.

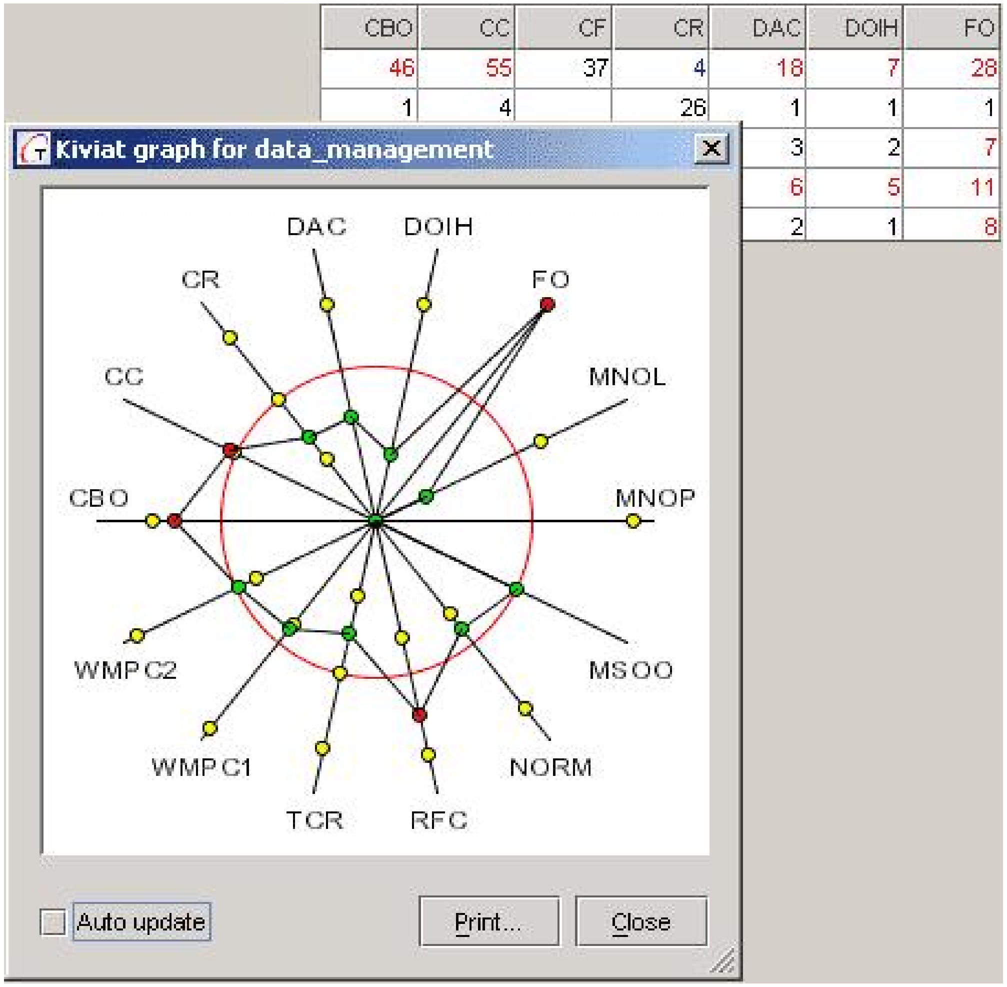
When you generate new metrics, the results will overwrite the original ones in the Message pane. Numbers in blue (CR in our picture) are lower than the lower limit. Numbers in red are higher than the upper limit.



The speedmenu for each cell of the metrics table has graph options. An interior cell (not in the first column or first row) has an option for both a Bar graph and a Kiviat graph.

Kiviat graphs correspond to the rows (classes and packages). Bar graphs correspond to the metrics.

For this step, put the cursor anywhere in the **data_management** row of the metrics table and select Kiviat graph from the speedmenu. Together will generate the graph, which shows the distribution of metrics over the package.



CBO	CC	CF	CR	DAC	DOIH	FO
46	55	37	4	18	7	28
1	4		26	1	1	1
				3	2	7
				6	5	11
				2	1	8

Tips and tricks

- Use audits and metrics as your first steps in refactoring code.
- Use Bar graphs when you want to consider the distribution of a particular metric over a collection of classes and packages.
- Use Kiviat graphs when you want to consider the distribution of different metrics over a particular class or package.

← PREVIOUS | TOP ↑ | NEXT → | START HOME

Last Revised: Tue, Apr 10, 2001

Together Tutorial
Part 11: Multi-User Support and Version Control

Together has multi-user version control that enables teams to share projects, diagrams, and source code. Together ships with CVS, and it can be easily integrated with any SCC-based version control system as well. (Indeed, Together's file-based architecture means that you can use your favorite file-based version control system client.)

This section will show how to put the airline project under version control.

Contents:

Putting a project under version control

Start this section by opening the airline project.

Step: Enable the airline project for version control.

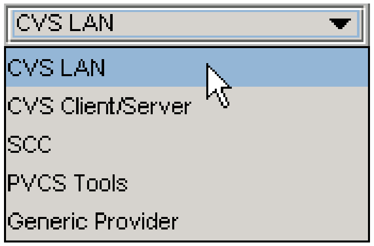
Putting a project under version control is a multi-step process. Begin with the project properties, available from the Main Menu under **File|Project Properties**.

1. If the Properties window does not show its "Resources" section, click on the Advanced button.

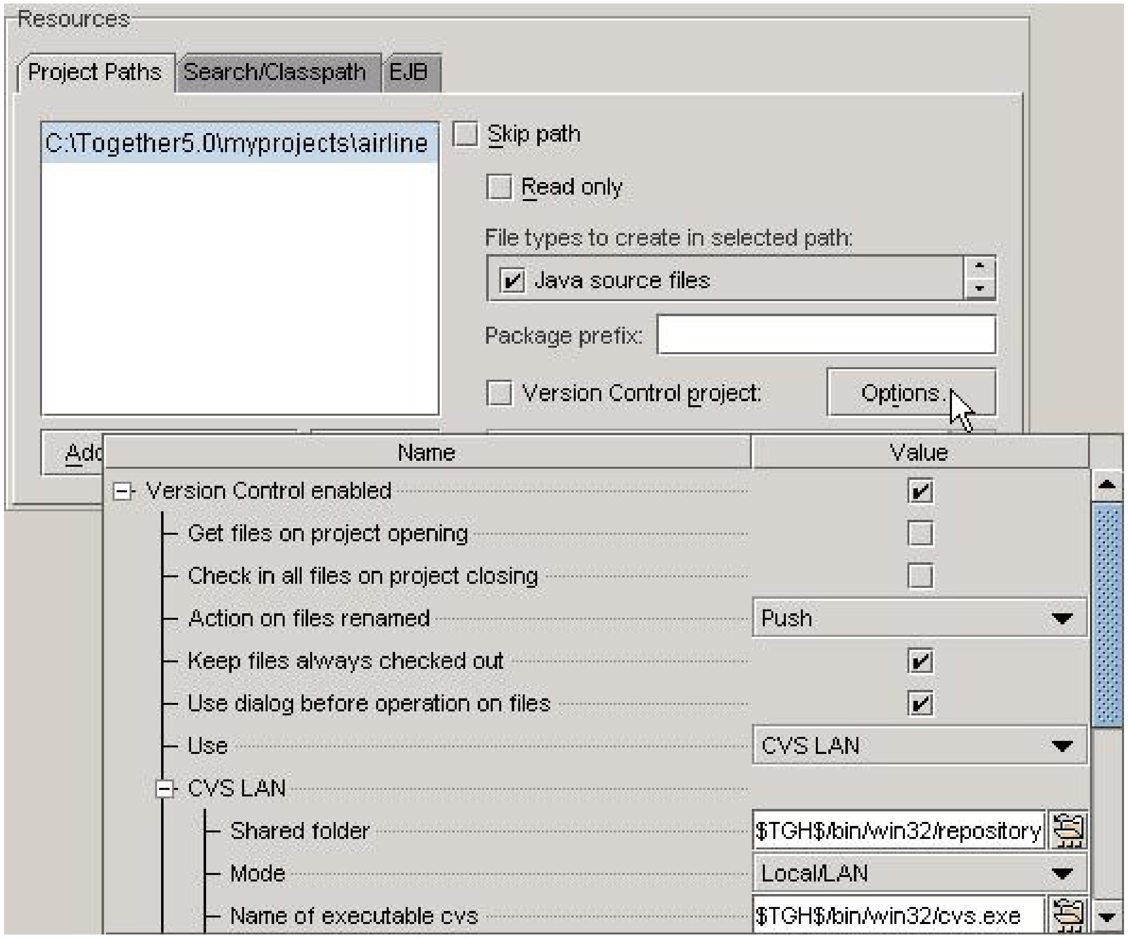
2. Click the Options button to open the Project Options window.

3. Check on **Version Control enabled** at the top of the Options window. Then expand **Version Control enabled** and scroll down to **Use** on the menu.

   (We used the default version control settings. Each version control system has its own menu.



4. Click OK on the Options window to return to the Properties window.



C:\Together5.0\myprojects\airline

□ Skip path

□ Read only

File types to create in selected path:

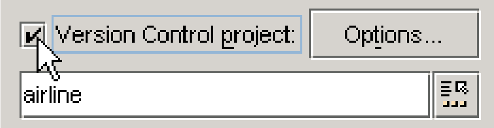
☑ Java source files

Package prefix:

□ Version Control project:

Name	Value
Version Control enabled	☑
Get files on project opening	□
Check in all files on project closing	□
Action on files renamed	Push
Keep files always checked out	☑
Use dialog before operation on files	☑
Use	CVS LAN
CVS LAN	
Shared folder	TGH/bin/win32/repository
Mode	Local/LAN
Name of executable cvs	TGH/bin/win32/cvs.exe

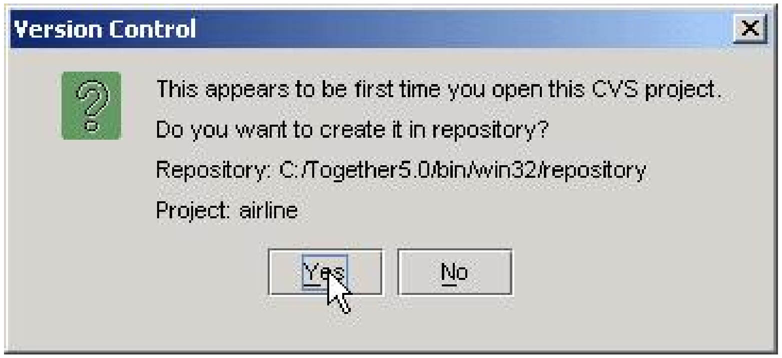
5. Check on Version Control project in the Properties window.



☑ Version Control project: Options...

airline

6. Click OK to close the Properties window.When you click OK to close the Properties window, Together will pop up a message box to create the repository. Click Yes.



Version Control

This appears to be first time you open this CVS project.

Do you want to create it in repository?

Repository: C:/Together5.0/bin/win32/repository

Project: airline

Adding files to version control

When you enable a project for version control, the class and diagram speedmenus show a new command, **Version Control**.

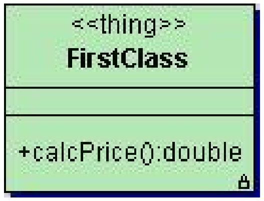
Step: Put **Agent.java**, **Ticket.java**, **Coach.java**, and **FirstClass.java** under version control. Do **not** keep them checked out.

You can add a single class to version control through **Version Control|Add** on its speedmenu. (When a class is not under version control, Add and System are the only available options.)

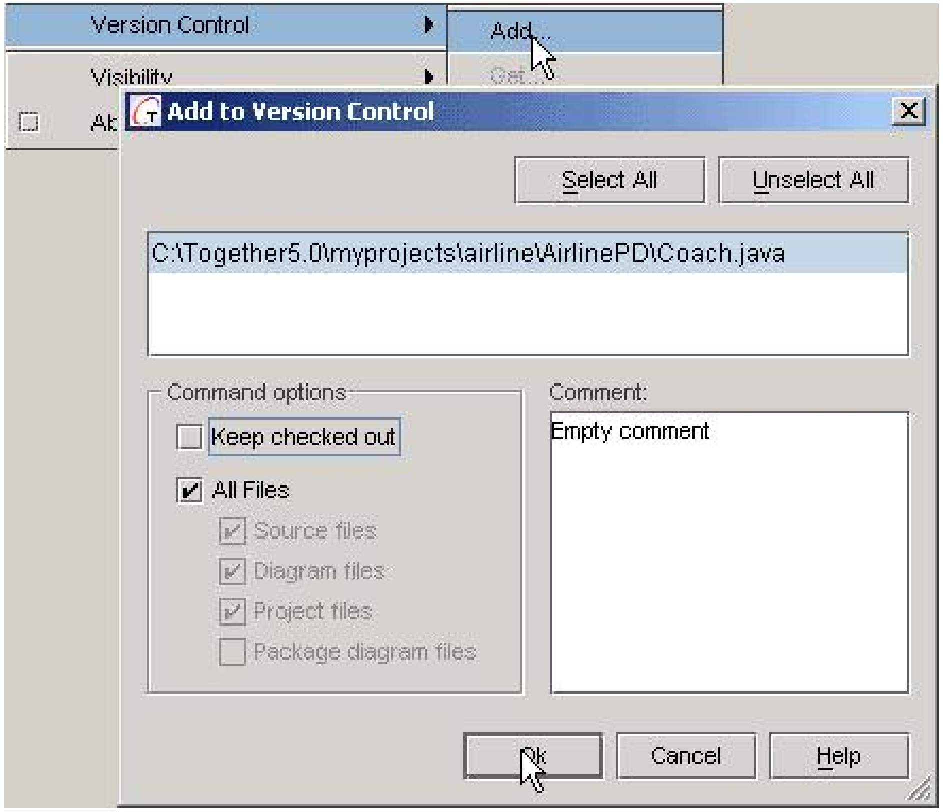
In the resulting dialog box, you should leave the first command option unchecked.

(If you want to add all class files, packages, or diagram files, you can go to the diagram speedmenu. You can also add via **Version Control|System** on any speedmenu.)

Together displays read-only files with a lock (🔒). Your **FirstClass** node of the **AirlinePD** package diagram should look like ours, with a lock in the lower right corner.



🔒 FirstClass The Explorer pane Model view shows a lock on **FirstClass** also.



Add to Version Control

Select All | Unselect All

C:\Together5.0\myprojects\airline\AirlinePD\Coach.java

Command options

☐ Keep checked out

☑ All Files

☑ Source files

☑ Diagram files

☑ Project files

☐ Package diagram files

Comment:

Empty comment

Ok | Cancel | Help

Checking files in and out

The files under version control should not be checked out prior to the following step. (If you neglected to uncheck the "Keep checked out" box in the **Add to Version Control** popup window, then the files you added will be checked out. In that case, select **Version Control|UnCheck out** from the class speedmenu.)

Step: Check out **Coach.java**. Then change the return statement in **calcPrice()** to: **return 199.0;**

You can check a file out through its speedmenu: **Version Control|Check out**. You'll get a dialog box similar to the one on the right.

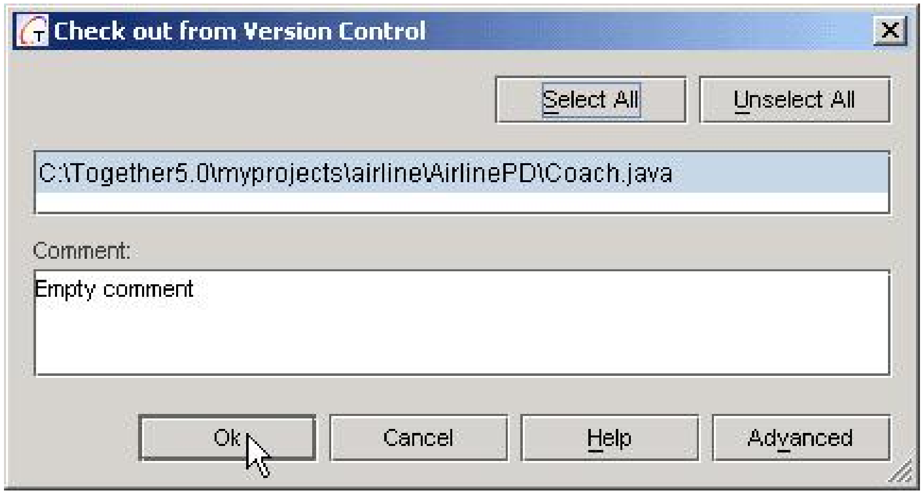
You can check out several files at the same time by lassoing them and going to the speedmenu. The lassoed files will appear on the upper (selection) panel.

All Version Control systems offer these options.

- Add -- adds a file to the version control system
- Get -- for looking at a file without changing it
- Check in -- returns the file to the repository
- Check out -- checks the file out of the repository

Together's dialog boxes vary according to which version control system is in effect. Under CVS LAN, Together has two additional options:

- UnCheck out -- to check a file back in without recording its changes since being checked out.
- Update -- to update the local copy of a file by merging in the other users' modifications from the repository.



C:\Together5.0\myprojects\airline\AirlinePD\Coach.java

Comment:

Empty comment

Step: Try to modify **FirstClass.java** the same way as you did **Coach.java**.
Then check **FirstClass.java** out. Modify it, changing the return statement in **calcPrice()** to: **return 499.0;**

A file under version that is not checked out is read-only. The Editor pane will not allow you to modify it.

Step: Check in **Coach.java**. Put a comment in the Comment portion of the Check-in dialog box.

You can check in a class or diagram via its speedmenu. The dialog box is similar to the Check-out dialog box.

We replaced "Empty comment" with "Correct coach class ticket pricing"

Examining version control system properties

You should now have four files under version control:

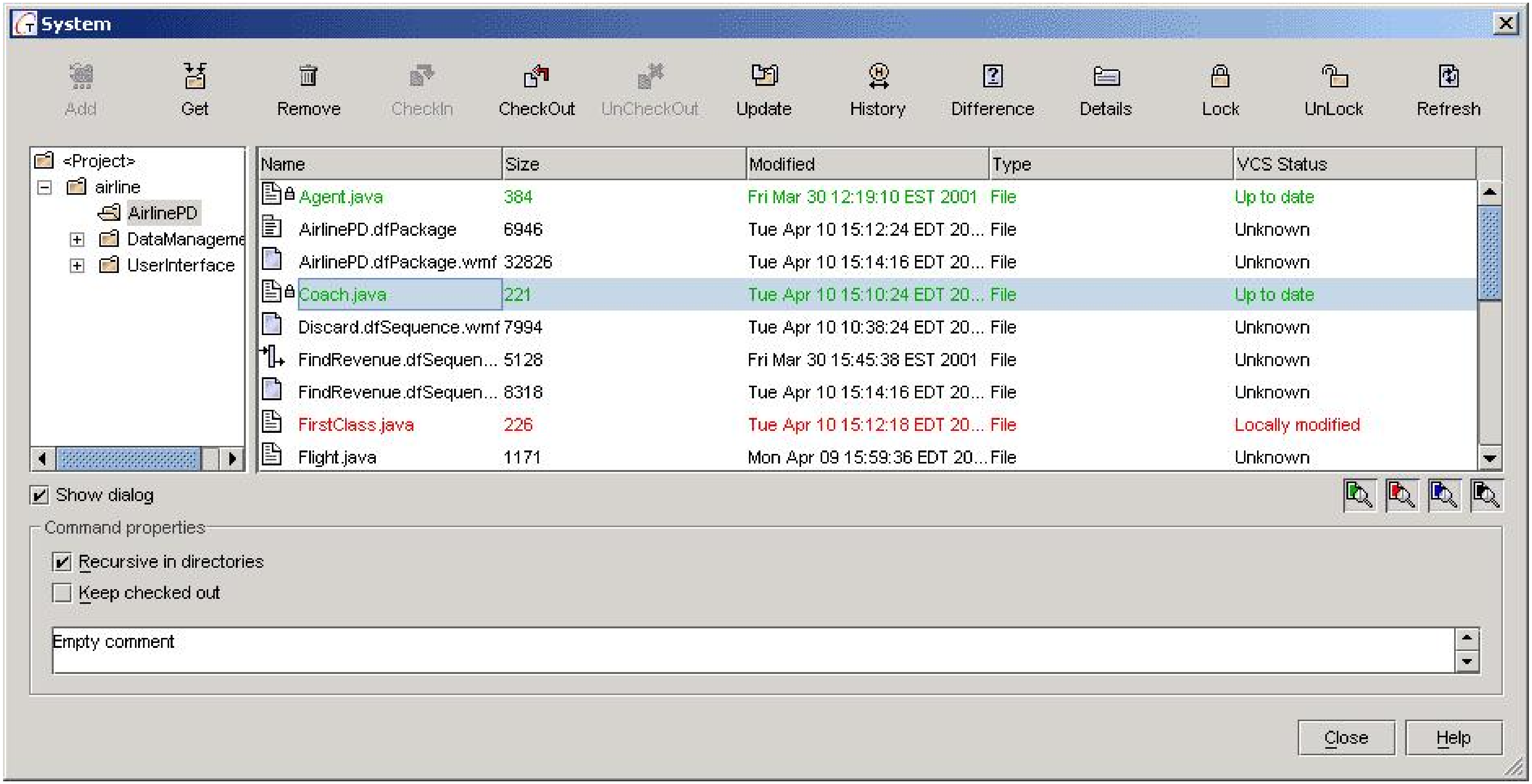
1. **Agent.java** never checked out.
2. **Ticket.java** never checked out.
3. **Coach.java** checked out, modified, and checked back in.
4. **FirstClass.java** modified and still checked out.

Step: Determine the status of each file via the Version Control System window.

This step is easy. The System window is a choice on the speedmenu for each class and diagram under version control.

Below is a picture of our System window. Notice the following:

- Files with black names are not under version control. You can click off the button under the file listing on the far right far right (🔍) to suppress listing them.
- **Agent.java** and **Coach.java** have read-only lock icons.
- **FirstClass** is listed in red, indicating that it has been modified.



Name	Size	Modified	Type	VCS Status
Agent.java	384	Fri Mar 30 12:19:10 EST 2001	File	Up to date
AirlinePD.dfPackage	6946	Tue Apr 10 15:12:24 EDT 20...	File	Unknown
AirlinePD.dfPackage.wmf	32826	Tue Apr 10 15:14:16 EDT 20...	File	Unknown
Coach.java	221	Tue Apr 10 15:10:24 EDT 20...	File	Up to date
Discard.dfSequence.wmf	7994	Tue Apr 10 10:38:24 EDT 20...	File	Unknown
FindRevenue.dfSequen...	5128	Fri Mar 30 15:45:38 EST 2001	File	Unknown
FindRevenue.dfSequen...	8318	Tue Apr 10 15:14:16 EDT 20...	File	Unknown
FirstClass.java	226	Tue Apr 10 15:12:18 EDT 20...	File	Locally modified
Flight.java	1171	Mon Apr 09 15:59:36 EDT 20...	File	Unknown

☑ Show dialog

Command properties

☑ Recursive in directories

□ Keep checked out

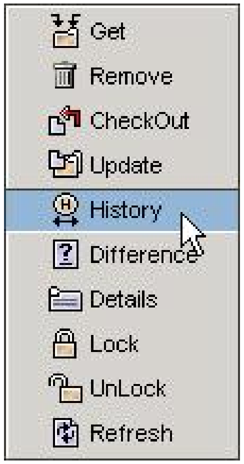
Empty comment

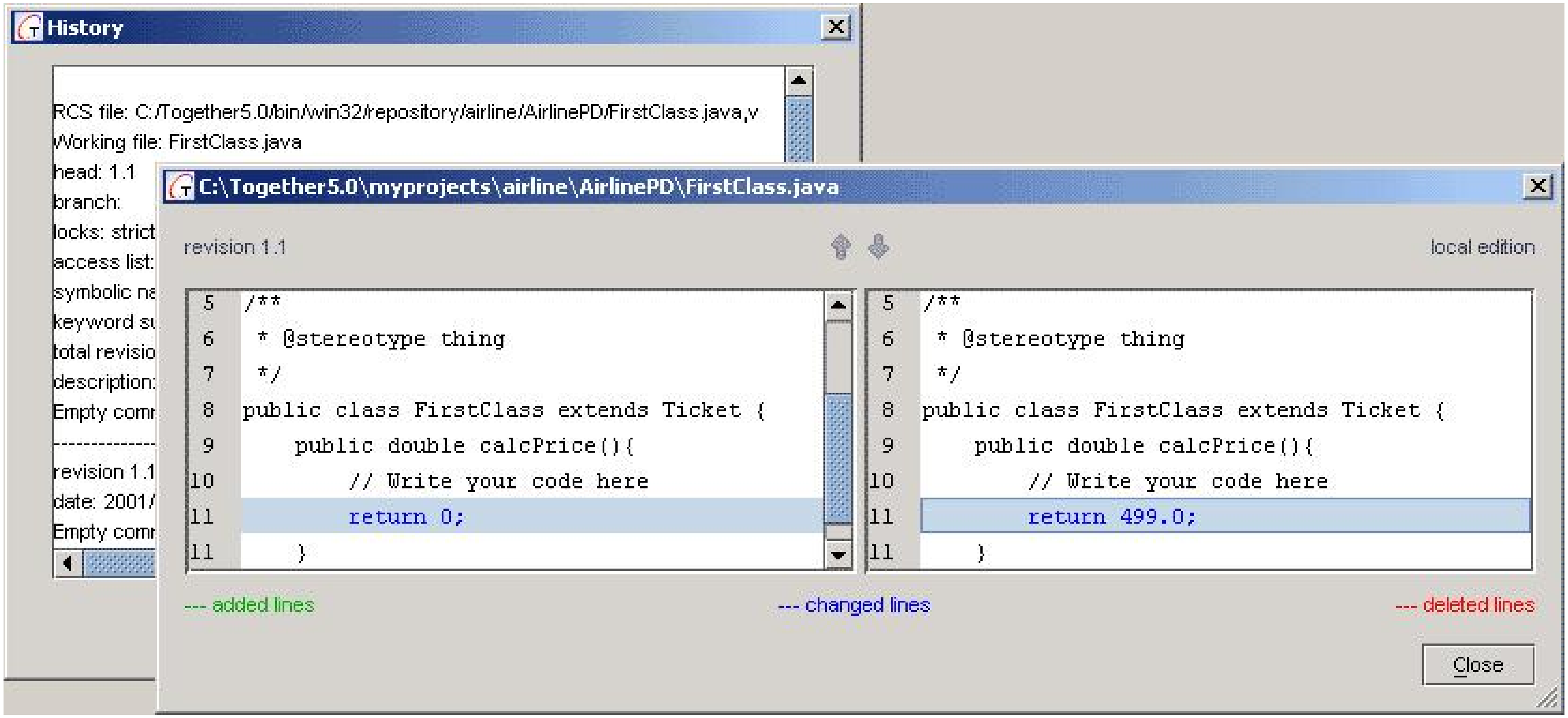
Step: Examine two items:

- the history of **Coach.java**
- the difference between **FirstClass.java** when it was checked out and now

Each item in the Version Control System window has a speedmenu. The items correspond to active items on main menu.

Shown below are the History and the Difference window for **FirstClass.java**. The Difference window displays original version of **FirstClass.java** on the left and modified version on the right. (History windows show comments.)





```
RCS file: C:/Together5.0/bin/win32/repository/airline/AirlinePD/FirstClass.java,v
Working file: FirstClass.java
head: 1.1
branch:
locks: strict
access list:
symbolic na
keyword su
total revisio
description:
Empty comr
-----------
revision 1.1
date: 2001/
Empty comr
```

```
/**
 * @stereotype thing
 */
public class FirstClass extends Ticket {
    public double calcPrice(){
        // Write your code here
        return 0;
    }
```

```
/**
 * @stereotype thing
 */
public class FirstClass extends Ticket {
    public double calcPrice(){
        // Write your code here
        return 499.0;
    }
```

Tips and Tricks

- Diagram files, source code files, and the project file (**.tpr**) typically go under version control. Workspace settings files (**.tws**) do not.
- Be careful using the Remove option for version control. It deletes the file.

← PREVIOUS | TOP ↑ | NEXT → | START HOME

Last Revised: Tue, Apr 10, 2001

Together Tutorial
Part 12: Running and Debugging Java Projects

In this final part of the Together Tutorial, you will learn how to use the Editor and the Message pane for compiling and debugging Java code. We have faithfully used the airline project for the first eleven parts of the Together Tutorial. But we will abandon it here in favor of the richer **CashSales** project.

On Windows platforms, Together installs and uses **javac.exe** from Java2^{TM} SDK version 1.3 as the default compiler. Unix users need to install the appropriate Java2 SDK and put it into the search path. With all platforms, Together provides a complete IDE for Java development.

Contents:

Viewing the Message pane as a simple console window

For a final bit of fun with the airline project, you'll add a simple output statement to the **Driver** class. (Remember that one? It's in the **UserInterface** package.)

Step: Put this statement inside **Driver.main()**.

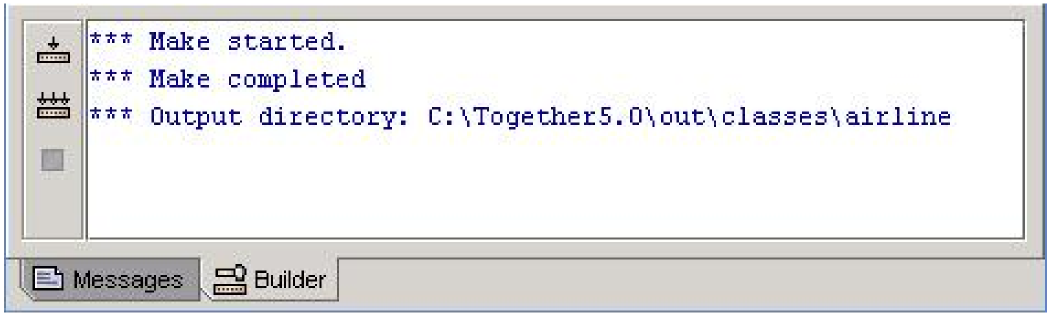
```
System.out.println("Goodbye Airline Project.");
```

Then compile the project.

Buttons for making () and building () a project are on the Main toolbar and on **Tools** from the Main menu.

Together will execute the default Java compiler and make utility, showing the results in the Message pane. If there are errors or warnings, you can click on the appropriate line inside the Message pane and navigate directly to the offending code.

You will see the results of compiling the project on the Message pane.



```
*** Make started.
*** Make completed
*** Output directory: C:\Together5.0\out\classes\airline
```

Step: Run the **airline** project.

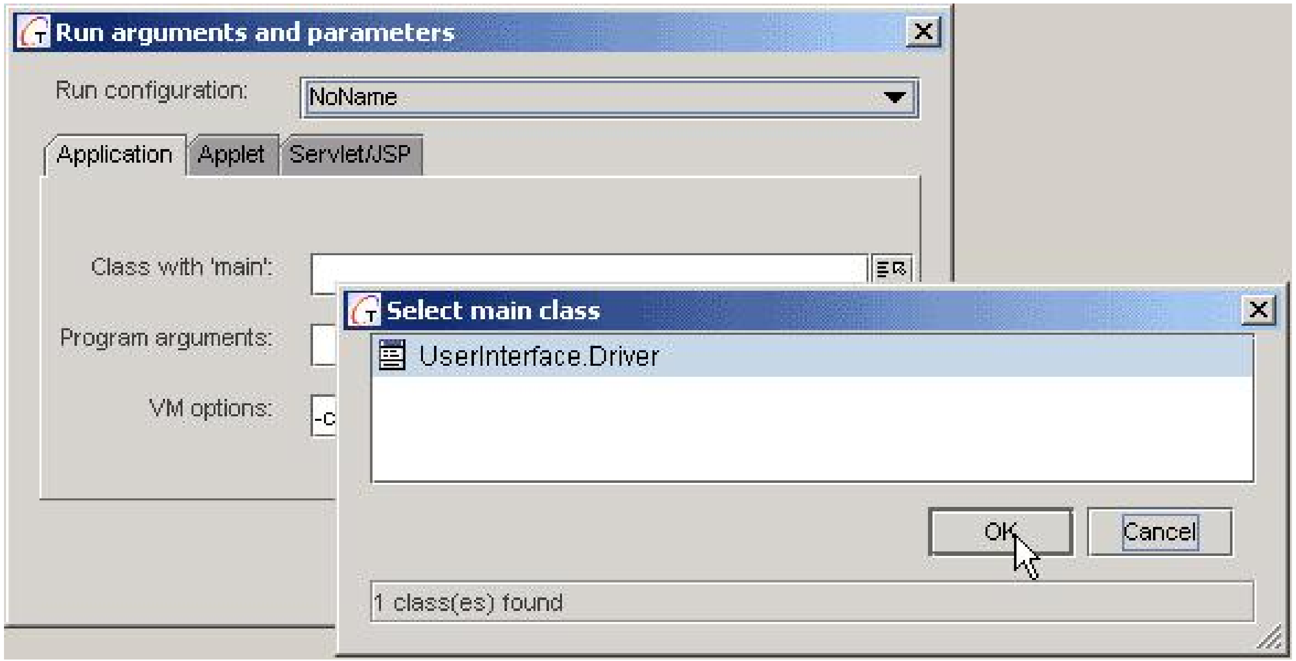
There's a run button (▷) on the Main toolbar also.

When you run a Java project, Together asks for the class with the **main** method. For this project, you have no choice. The snapshot on the right shows the pop-up windows from the run command.

The Message pane gives Together a simple console window. The Message pane is the focus of standard input and output of programs running within Together.

Shown below is the resulting execution. The tab at the right of the pane displays the class with the main.

Notice that the Message pane no longer has a single tab. You can close a tab with the tab speedmenu.



Run arguments and parameters

Run configuration: NoName

Application | Applet | Servlet/JSP

Class with 'main':

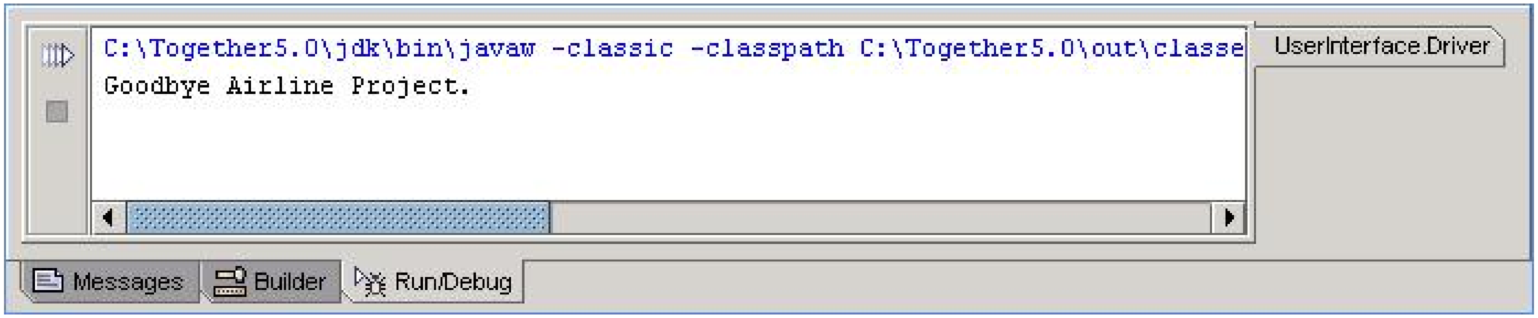
Program arguments:

VM options: -c

Select main class

UserInterface.Driver

1 class(es) found



```
C:\Together5.0\jdk\bin\javaw -classic -classpath C:\Together5.0\out\classe
Goodbye Airline Project.
```

This wraps up all of the Together Tutorial work on the airline project. Congratulations. You've learned a lot!

Setting breakpoints and animating the debugger

Step: Open the CashSales project.

Opening a project ought to be easy. There are several options.

- Use the **File|Open Project** command.
- If **CashSales** has been opened recently, use the **File|Reopen** command.
- From the Directory tab of the Explorer pane, click on **CashSales.tpr** in the **CashSales** folder.

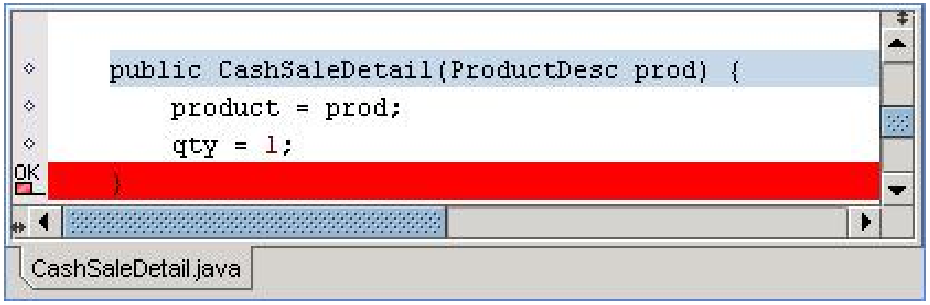
Together warns you that it cannot open another project without closing the current one.

Step: Set a breakpoint at the last curly brace in the constructor for **CashSaleDetail** (from the **problem_domain** package).

You need to pull up the code in the Together editor. Having trouble getting there? Use the Overview tab in the Explorer to steer **CashSaleDetail** to the middle of the Diagram pane... and go from there.

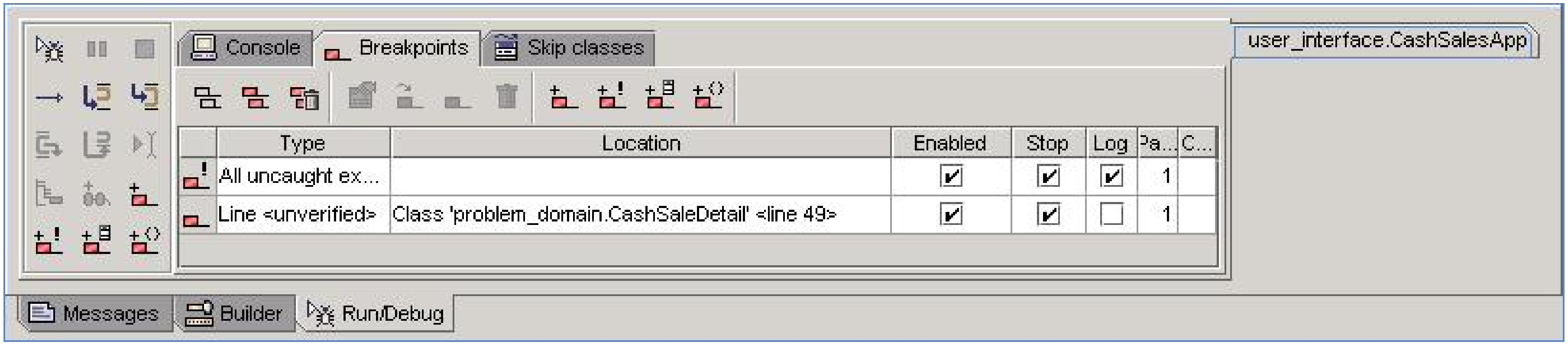
Once you find the line in the editor, click the cursor at the line on the Editor's left margin. This will highlight the line in red and put a breakpoint icon at the left margin.

Mouse clicking on the left margin toggles adding and removing the breakpoint.



```
    public CashSaleDetail(ProductDesc prod) {
        product = prod;
        qty = 1;
OK  }
```

The Debugger tab of the Message pane displays all of the breakpoints. (If the tab is not showing, select **Tools|Run/Debug|Run|Show debugger tab** from the Main menu.)

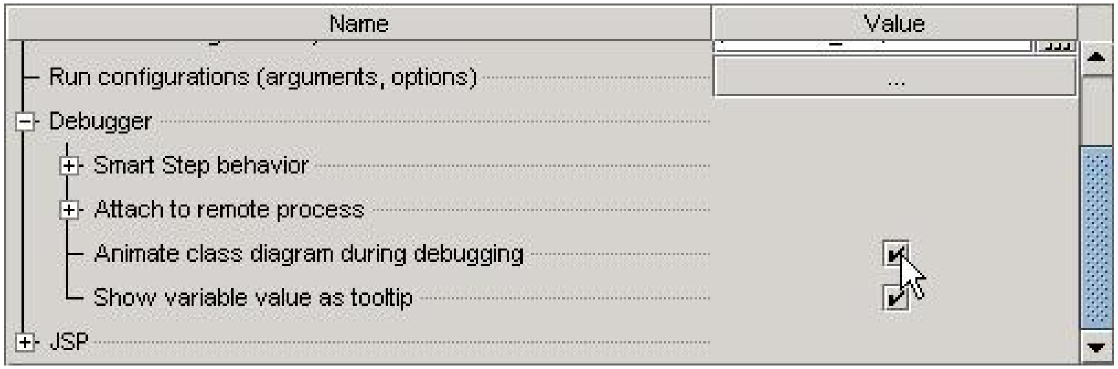


	Type	Location	Enabled	Stop	Log	Pa...	C...
	All uncaught ex...		☑	☑	☑	1	
	Line <unverified>	Class 'problem_domain.CashSaleDetail' <line 49>	☑	☑	□	1	

Step: Set the debugger to show the connection between diagram and code while it is running a project.

Go to the Debug tab on the Project or Default Options (available on **Options** from the Main menu). Check **Display class/member in class diagram** on.

Checking on this option has the effect of animating the debugger. When you run the project in the animated debugger, Together will scroll the class diagram to highlight the method that you are stepping through.



Name	Value
Run configurations (arguments, options)	...
⊟ Debugger	
⊞ Smart Step behavior	
⊞ Attach to remote process	
Animate class diagram during debugging	☑
Show variable value as tooltip	☑
⊞ JSP	

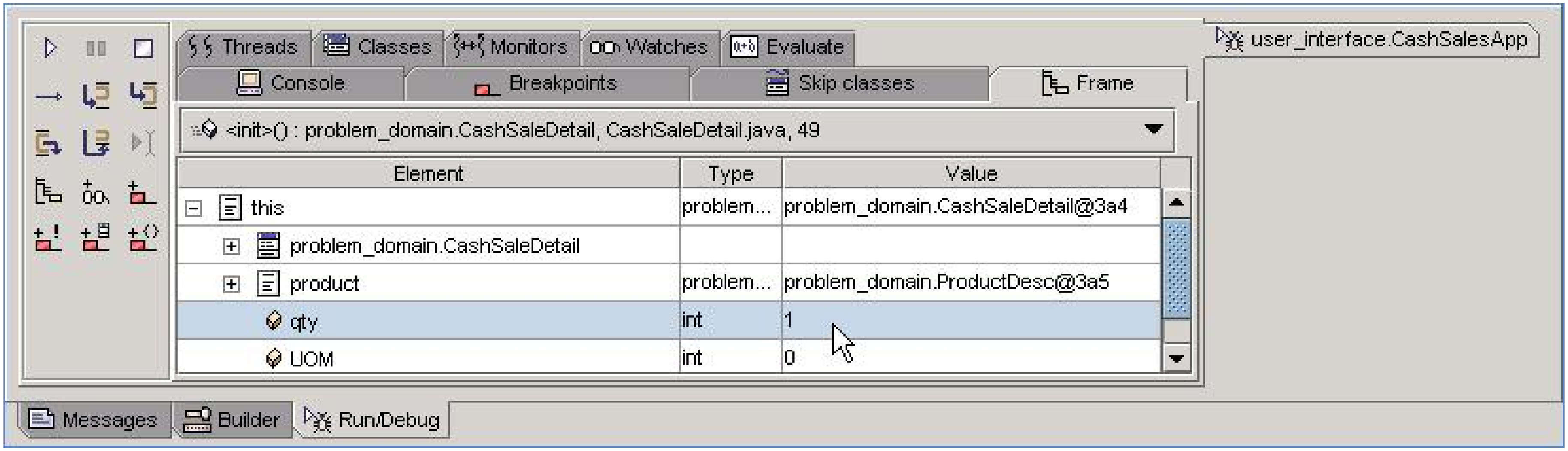
Running under the debugger

Step: Run the **CashSales** application under the debugger. At the breakpoint, examine the current stack frame and the value of **CashSaleDetail**.

The Main toolbar has a Run in Debugger button (). When debugging execution begins, the Debugger tab expands to show detailed runtime information. (Be patient for the application to get going. You'll need to press the "Scan" button to get to the breakpoint.)

The snapshot below was taken while execution of **CashSales** was suspended at the breakpoint. The left side of the Debugger pane shows a toolbar, with buttons to guide execution of the program, examine frames and threads, and set various kinds of breakpoints.

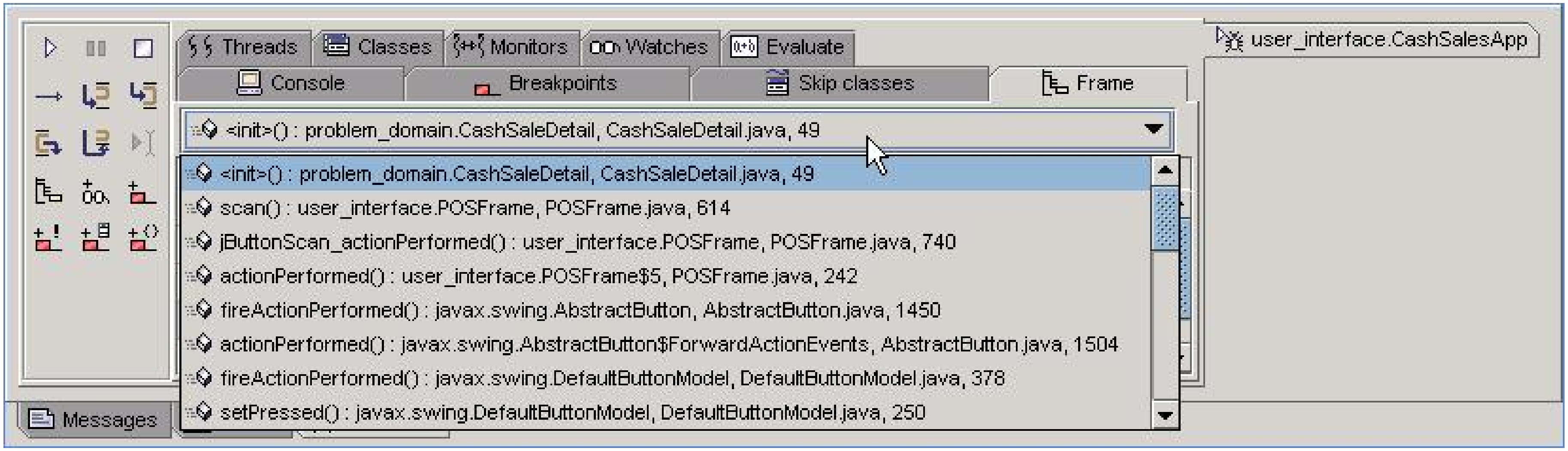
The Frame tab (at the front in the snapshot) shows the contents of the current stack frame when it hits the breakpoint. At that point, you can expand **this** to see **CashSaleDetail.qty**.



<init>() : problem_domain.CashSaleDetail, CashSaleDetail.java, 49

Element	Type	Value
⊟ this	problem...	problem_domain.CashSaleDetail@3a4
⊞ problem_domain.CashSaleDetail		
⊞ product	problem...	problem_domain.ProductDesc@3a5
qty	int	1
UOM	int	0

Clicking the method name at the top of the Frame tab shows the runtime stack for this thread.



```
<init>() : problem_domain.CashSaleDetail, CashSaleDetail.java, 49
<init>() : problem_domain.CashSaleDetail, CashSaleDetail.java, 49
scan() : user_interface.POSFrame, POSFrame.java, 614
jButtonScan_actionPerformed() : user_interface.POSFrame, POSFrame.java, 740
actionPerformed() : user_interface.POSFrame$5, POSFrame.java, 242
fireActionPerformed() : javax.swing.AbstractButton, AbstractButton.java, 1450
actionPerformed() : javax.swing.AbstractButton$ForwardActionEvents, AbstractButton.java, 1504
fireActionPerformed() : javax.swing.DefaultButtonModel, DefaultButtonModel.java, 378
setPressed() : javax.swing.DefaultButtonModel, DefaultButtonModel.java, 250
```

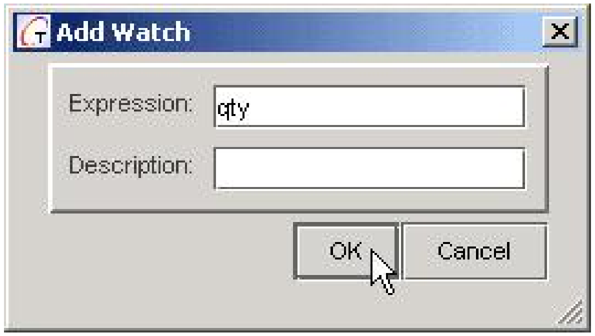
The left margin of the Editor changes when a project is running under debug mode. Small diamonds (◇) mark executable lines. And the breakpoint icon changes from not running (▬) to running (OK).

Setting watchpoints and changing program execution

Step: Set a watchpoint on **CashSaleDetail.qty**. Then change the value of **qty** to **47**.

You can set a watchpoint when execution is suspended at a breakpoint. Click the watchpoint icon (OO) on the leftmost panel of the Message pane.

You'll get a dialog box similar to the one on the right. Enter the expression as shown.

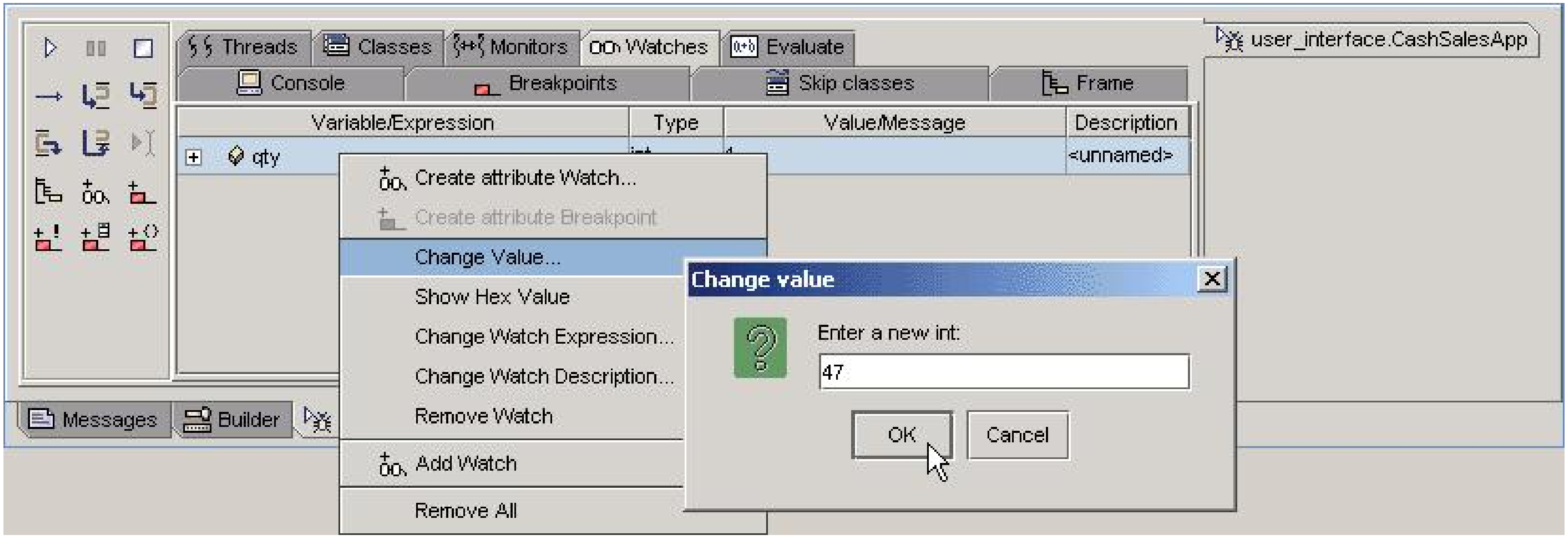


Expression: qty

Description:

You can set the value of a watchpoint expression with the pull-down menu on the watchpoint line. First, left click the line to highlight it in blue. Then right click to get the menu. For this exercise, select "Change Value."

You can set the value of qty in the resulting dialog box.



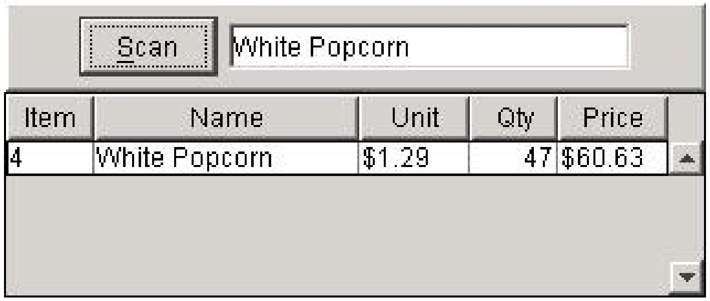
Variable/Expression	Type	Value/Message	Description
qty	int	[illegible]	<unnamed>

Change value

Enter a new int: 47

Click the continue button (▷) on the Message pane to get going again.

The snapshot to the right shows the result on our user interface. The result will vary according to the item scanned.



White Popcorn

Item	Name	Unit	Qty	Price
4	White Popcorn	$1.29	47	$60.63

Tips and Tricks

- Don't use the make button on a project that is up to date or that has nothing to build. You can force a make by choosing **Tools|Rebuild Project** from the Main menu.
- You can remove a breakpoint via its speedmenu in the Debugger tab. Select the breakpoint from the Debugger list and right click. There's a "Remove breakpoint" option. Alternatively, you can select the breakpoint and hit Delete.
- When execution is suspended at a method call, clicking on the "Step into" button (↳) will not step into the method when Smart Step is on. (The Debugger tab has a "Skip classes" tab that lists the classes that the debugger won't enter. By default, these are classes in the standard Java class libraries. You can add more classes to the list with the "Skip classes" speedmenu. The tab has a check box that allows you to step into the class, assuming its source code is available.)
- C++ programmers can compile their code within Together. But first, they need to modify the compiler specification in the Together configuration options to point to their C++ compiler.

← PREVIOUS | TOP ↑ | START HOME

Last Revised: Thu, Apr 12, 2001